SOFTWARE VULNERABILITY REPAIR

RIDWAN SALIHIN SHARIFFDEEN

NATIONAL UNIVERSITY OF SINGAPORE

2022

SOFTWARE VULNERABILITY REPAIR

RIDWAN SALIHIN SHARIFFDEEN

(*B.Sc. Engineering (Honours), University of Moratuwa, Sri Lanka*)

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

SCHOOL OF COMPUTING

NATIONAL UNIVERSITY OF SINGAPORE

2022

Advisor:
Professor Abhik Roychoudhury

Examiners:
Associate Professor Khoo Siau Cheng
Associate Professor Liang Zhenkai

DECLARATION

I hereby declare that this thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis.

This thesis has also not been submitted for any degree in any university previously.

Ridwan Salihin Shariffdeen

November 2022

To the tax-payers who supported and contributed to free education in Sri Lanka!

ACKNOWLEDGEMENTS

Four and half years of my PhD journey was a roller-coaster ride with many ups and downs, that helped me to be a better researcher and transformed me to become a better person. My experience pushed my limits both mentally and physically, which helped me to discover my strengths and weaknesses. I would not have come this far, without the tremendous support from my family, advisors, friends and colleagues.

First and foremost gratitude goes to my PhD advisor Professor Abhik Roychoudhury for his continuous and rigorous support in the past several years. Abhik taught me many important aspects in research, critical thinking, idea exploration, solution presentation and influential work. His guidance on first-principle thinking is the most valuable that helped me in my research career, not only for the problems in our community but also to be able to appreciate contributions in other fields. Through his patient guidance and valuable directions, I have achieved more than I ever could.

I would like to take this opportunity to thank my thesis advisory committee, Professor Liang Zhenkai and Professor Siau-Cheng Khoo for their valuable feedback, comments and support. I would also like to extend my appreciation to the exceptional minds, with whom I had the privilege to work as a collaborator, Yannic Noller, Gregory J Duck, Gao Xiang, Shin Hwei Tan, Lars Grunske, Julia Lawall, Jiaqi Tan and Mingyuan Gao. It was a great experience working with them, discussing research questions and exploring ideas. Without them, I would not have achieve much alone.

I would specially thank Sergey Mechtaev, Shin Hwei Tan and Gao Xiang for not only helping me during my initial stages in the PhD but also for the inspiration they gave to work on program repair. I was also fortunate to work with some wonderful souls in our lab: Zhen Dong, Yannic Noller, Gao Xiang, Yuntong Zhang, Dylan Wolff, Ruijie Meng, and Andreea Costea. I truly enjoyed our discussions on various aspects of life as a researcher providing different perspectives.

My PhD would have not been a success without the support and encouragement I got from my friends, especially during the arduous times when I needed the most. Sumudu Herath, Manupa Karunaratne, Ashan Asmone, Sunimal Rathnayake, Dileepa Fernando, Rajith Vidanarachchi, Rajith Gun Hewage, Dhananjaya

Wijerathne and Viraj Herath. Your word of encouragement, imparting wisdom and tremendous support helped me hustle through the grinding moments in my PhD journey. Special thanks for Sanka Rasnayaka for encouraging me to join the PhD program at NUS.

Apart from the academic support, I had enormous support for many other aspects, during the years living in Singapore. Especially the Sri Lankan community who made me feel home despite living in a foreign country. Lihil Subasinghe, Thushara Sandakalum, Sandareka Wickramanayake, Kalpa Welivitigoda, Supuli Jayaweera, Gayana Herath, Shashikala Perera, Udari De Alwis, Dumith Jayathilaka and Nuwan Janaka. Thank you for all the memories we created.

I would also like to mention my friends from the January 2018 PhD intake, with whom I had the privilege of sharing the journey together. Rruba Panchendrarajan, Suman Panda, Neamul Kabir, Sylvain Riondet, Rafael Peres, Muhammad Rizki, Martin Strobel and Xiao Liang Yu. Thank you for your camaraderie.

I am also grateful for those who helped me in my early career, my undergraduate advisor Dilum Bandara, my mentors Zanfer Hussain and Thilanka Kaushalya. Thank you for supporting me in my work, providing valuable advice and giving me the encouragement to pursue a career in research.

Above all, although this alone is not sufficient, I would like to thank my family for their endless patience and understanding throughout all the years. Thank you for your continuous support and endless love!

I am extremely grateful for the financial support provided by the National Satellite of Excellence in Trustworthy Software Systems and to the National Research Foundation, Prime Minister's Office, Singapore for its Campus for Research Excellence and Technological Enterprise (CREATE) programme.

None of this would have been possible if not for the teachers who taught me the ABC's and guided me in the path for higher-education. Remembering the dear teachers at Vidyalankara Kanishta Vidyalaya Negombo, St. Anthony's College Wattala and University of Moratuwa.

I am eternally grateful to the people of Sri Lanka and Singapore, for supporting and sponsoring my education which paved the path to this thesis.

Contents

Abstract

Automatically generating security patches provides proactive protection against exploitation of known vulnerabilities by malicious users. The problem of automatic patching is a long-standing requirement in practice, which is not limited to generating a fix for the identified software security vulnerability but may also require generalizing the fix to the extent that it can be ported to other similar variants of the vulnerability that exist in different software systems.

This thesis introduces a series of cohesive techniques tightly coupled towards the goal of generating security patches for identified software security vulnerabilities. First, we study the impeding challenges in trusted program repair, specifically addressing the trustworthiness of auto-generated patches. Considering the insights gained from our study, we propose "compilation-free program repair" to speedup the efficiency of program repair. Third, we propose a novel program repair technique “concolic program repair” that integrates a user-provided program-specification to guide program repair to find the correct patch while efficiently navigating a large search-space. In doing so, we also provide additional guarantees for the correctness of the generated patches by generating additional test-cases. Fourth, inspired by program synthesis technique, we propose a novel transformation rule synthesis algorithm that can produce properly generalized transformation rules to automatically backport trusted patches to older versions of the same software. Last, we propose a code transplantation technique to repair semantically equivalent programs that exhibit potential for a similar variant of the identified vulnerability.

We perform a comprehensive set of experiments on reported software security vulnerabilities in real-world applications inclusive of the Linux kernel project, subjects from Google’s Open-source-systems (OSS) Fuzz framework and other popular large-scale software applications. Our experiments showed that the proposed techniques advance the state of the art program repair to address the challenges in generating security patches for software security vulnerabilities. Our proposed techniques should serve a long-standing need in practice.

List of Tables

List of Figures

Publications Appeared

Following lists the publications that have been created along this thesis work and that served as basis for this thesis.

- **Ridwan Shariffdeen**, Shin Hwei Tan, Mingyuan Gao, Abhik Roychoudhury. *"Automated Patch Transplantation"*. ACM Transactions on Software Engineering and Methodology (TOSEM), 30(1), pages 1-36, 2021.

- **Ridwan Shariffdeen**, Xiang Gao, Gregory J Duck, Shin Hwei Tan, Julia Lawall, Abhik Roychoudhury. *"Automated Patch Backporting in Linux (Experience Paper)"*. In the Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA) 2021. [joint first-author with Xiang Gao]

- **Ridwan Shariffdeen**, Yannic Noller, Lars Grunske, Abhik Roychoudhury. *"Concolic Program Repair"*. In the Proceedings of the 42nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI) 2021. [joint first-author with Yannic Noller]

- Yannic Noller, **Ridwan Shariffdeen**, Xiang Gao, Abhik Roychoudhury. *"Trust Enhancement Issues in Program Repair"*. In the Proceedings of the IEEE/ACM 44th International Conference on Software Engineering (ICSE) 2022. [joint first-author with Yannic Noller]

- **Ridwan Shariffdeen**, Gregory J Duck, Jiaqi Tan, Abhik Roychoudhury. *"Compilation-free Program Repair"*. In the Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE) 2022. [submitted]

Chapter 1

Introduction

Software developers must often face the unpleasant discoveries of unexpected or undesirable behaviors known as bugs. One inevitable side-effect of such bugs is the existence of vulnerabilities that can be exploited by malicious users for their own advantage e.g. disclose confidential information. Software vulnerabilities pose serious security risks to computer systems. *Heartbleed* vulnerability (CVE-2014-0160) is an infamous example of such a software vulnerability that could lead to disclosure of passwords and private keys handled by applications that use OpenSSL [32]. A special characteristic of a software security vulnerability is the wider range of applicability in the software ecosystem. Furthermore, current practices in fixing software security vulnerabilities involve providing a patch to the latest version(s) of the software and require the users to upgrade, letting thousands of dependent packages of older versions exposed to the vulnerability [28]. Due to various reasons such as dependency-locks in software systems, there exists a barrier to attain the correct fix to some versions of the software. For example, the search for Heartbleed vulnerable devices conducted by Shodan in March 2016 [124], returned 237,539 results indicating many devices remained affected by this vulnerability years after a publicly available patch was provided. Most of the affected devices that were reported as vulnerable, used old versions of OpenSSL (1.0.1 - 1.0.1f), this may be due to version lock-in. In order to prevent this vulnerability exploitation, users were requested to upgrade to the latest version of OpenSSL in which the issue was fixed. But, upgrading a dependent software is not a trivial task and in some cases it can be difficult due to tight-coupling of the specific version in use.

A vulnerability in a software system poses a significant threat to the security and reliability not only for the said software but also for the whole software ecosystem.

The same vulnerability can exist in other forms and facades in different systems due to the reuse of source code, libraries and artifacts at higher levels of abstraction (i.e. JPEG 2000 standard, SSL/TLS protocols). With the advent of component-oriented programming, the security issues of code reuse is an important aspect that requires analysis as well as maintenance strategies to preserve the security of all systems, not just the software where the vulnerability was detected. One of the recently disclosed vulnerabilities that affected almost all Linux, iOS, and Android devices is the KRACK [130] vulnerability. Even though the original program was fixed, dependent software required days to merge the upstream fix due to variances between the original program and the integrated program [129]. Hence, any attacker who was exploiting the KRACK vulnerability had an advantage of a few days to launch their attacks targeting not-yet-patched systems.

Debugging is a costly and time-consuming activity, especially when considering large projects of thousands of lines that have been developed for years and by several programmers. Furthermore, even when the origin of a bug is correctly identified, a fix can be nontrivial. Hence, fixing an existing security vulnerability itself is often a difficult task that requires a deep understanding of the semantics, context, and environment of the application [121]. Another challenge in fixing software security vulnerabilities is the propagation of the fix to prevent all similar variants of the vulnerability that can be exploited. Due to the intertwined nature of a patch to a software security vulnerability and the vulnerability itself, the existence of a patch reveals the existence of a vulnerability. Hence, any un-patched version of the software is exposed to be targeted by malicious users. A cohesive set of techniques are required in-order to completely mitigate the threat of an identified software security vulnerability. More precisely, once a software security vulnerability is identified, all software that is affected by the defect should be repaired at the same time or prior to the publication of the existence of the vulnerability.

The tedious task of fixing bugs is not limited to the security domain. To relieve the burden of fixing bugs, many techniques have been previously proposed for automated program repair such as genetic programming [68, 155], semantic analysis based repair techniques [92, 105, 137, 91, 161, 138, 136, 78], machine learning based techniques [79] and hybrid approaches [69]. Although existing automated repair techniques are capable of fixing some of the software errors in real-world applications, these techniques are often ineffective in patching software security vulnerabilities [138, 110].

Furthermore, once a patch is discovered that can fix an instance of the identified software security vulnerability, the task remains to generalize the patch and apply to all other instances of the vulnerability. In practice, developers manually port [111, 112] fixes selectively from one program to another program. The program can be of the same project in different versions or completely different projects that share a higher-abstraction such as a standard or a protocol. When porting code from one context to another, semantics of the ported code often change due to differences in the surrounding contexts. Developers may overlook such subtle differences, inadvertently creating a porting error [112, 111].

This thesis focuses on alleviating the problem of generating patches for security vulnerabilities, which can help developers to quickly patch identified security vulnerabilities and reduce the exposure time for exploitation not only for the instance that was reported but to other similar variants of the identified vulnerability.

1.1 Motivation

The security of computer systems are severely affected by the existence of vulnerabilities and threaten the IT infrastructure of many organizations. Among different kind of vulnerabilities, certain vulnerabilities affect more than one system, where the issue lies in a shared library, incorrect implementation of a protocol or simply an error in a standard itself. For instance, previously disclosed *Heartbleed* [128] and *Shellshock* [131] vulnerabilities that affected almost any software that used OpenSSL libraries and a vulnerability in the server message block (SMB) protocol exploited by the *WannaCry* ransomware [24] have affected a wide range of systems and millions of users worldwide. One of the recently disclosed security issues known as *ZipSlip* [132] is a widespread critical archive extraction vulnerability, allowing attackers to write arbitrary files on the system, typically resulting in remote command execution. The vulnerability has been found in multiple ecosystems, including JavaScript, Ruby, .NET and Go, but is especially prevalent in Java, where there is no central library offering high-level processing of archive (e.g. zip) files. The lack of such a library led to vulnerable code snippets being hand-crafted and shared among developer communities such as StackOverflow. *KRACK* [130] is another instance, where the protocol itself has flaws in its implementation and lead to one of the deadliest security issue found in recent times. The weaknesses are in the Wi-Fi standard itself, and not in individual products or implementations. Therefore, any correct implementation of WPA2 was affected. To prevent the attack, users were forced to update affected products as soon as security updates become available. During the initial research, it was discovered that Android, Linux, Apple, Windows, OpenBSD, MediaTek, Linksys, and others, are all affected by some variant of the attacks. Even though the original patch was released in September 2017, it took couple of weeks for many vendors to integrate the patch into their stack [129], providing enough exposure window for malicious users to target such instances.

Despite recent advancements in detecting software security vulnerabilities [106, 56, 14, 107, 108], attacks against computer software are reported on an almost daily basis. Software systems that are developed, deployed and used over years contain significant security weaknesses, where over 90% of security incidents reported are from software defects [149]. According to [27] and [162], one of the major causes of security incidents and breaches can be attributed to the exploitable vulnerabilities in software. Once a vulnerability is exploited by attackers, companies and organizations may suffer from significant financial loss as well as irreparable damage to their reputation [123]. Delay to fix such issues have caused significant cost up to 200 times as much as early correction [12]. For example, one of the recently disclosed and widely exploited vulnerabilities known as *EternalBlue* actually involved CVE-2017-0143 to 48, a family of vulnerabilities related to the Microsoft SMBv1 server protocol that was used in certain Windows versions. The infamous *WannaCry* ransomware outbreak in 2017 exploited this N-day vulnerability in unpatched computers. Interestingly, 3 years after the first outbreak, *WannaCry* was still the most detected malware family in 2020, and according to Shodan, more than 650,000 internet devices remain vulnerable to it. The same vulnerability was influential behind the *NotPetya* cyberattack across the world, which caused more than US$ 10 billion in estimated damage [150]. More recently, one of the most severe vulnerability reported in Log4j, an open-source logging library commonly used by thousands of apps and services across the internet and its part of the Apache

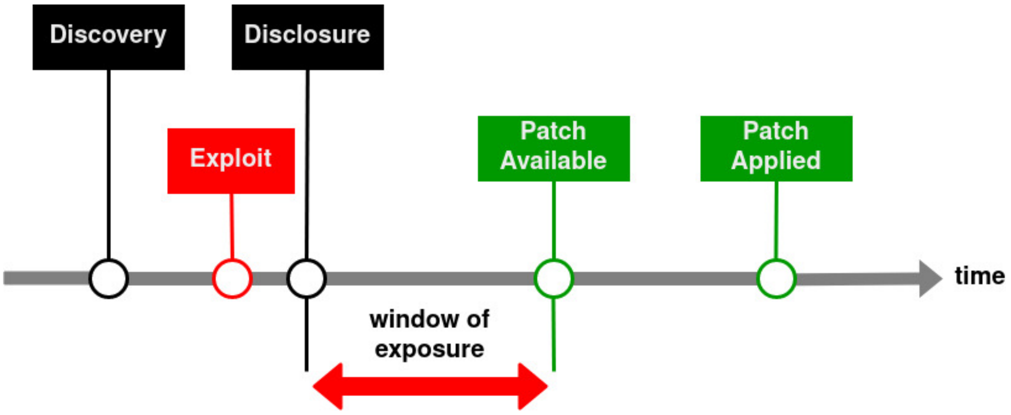
Logging Services, a project of the Apache Software Foundation. The critical vulnerability exposed applications to a remote-code-execution attack[49]. Within just a few hours, vast number of attempts at exploiting Log4j vulnerabilities(s) were reported by several threat monitoring services, indicating malicious users are quick to learn and attack at newly found vulnerabilities.

It is important to repair identified vulnerabilities as quickly as possible and take the necessary steps to minimize the impact. One of the crucial steps towards defense against identified vulnerabilities is to integrate security patches into one's system as quickly as possible.

1.2 Problem Formulation

Figure 1.1 depicts the life-cycle of a vulnerability from the early stages of discovery to the later stages of remedying by applying a patch. Two distinctive stages in the life-cycle of a vulnerability are important for our work, the disclosure time and the patch time (i.e. the time of patch availability). Note that the sequence of the exploit time, disclosure time and patch time are not fixed. This thesis aims to reduced the window of exposure by automatically generating patches for software security vulnerabilities.

Figure 1.1: Life-cycle of a typical vulnerability



Distinctive stages in time divide the life-cycle of a vulnerability into several phases.

The disclosure time of a vulnerability is defined as the first date information of the vulnerability is publicly released. Similarly, the time of a patch availability is defined as the first date the vendor of the software releases a patch that provides protection against the exploitation of the vulnerability. The time duration where the vulnerability is disclosed and a patch is released by the vendor is known as the `window of exposure`. The window of exposure is where the developers of the software has control and responsibility, hence the goal of the vendor is to minimize this window to zero days. Thus to minimize the impact of the vulnerability, a patch should be released by the vendor on the same day where the vulnerability is disclosed.

While software vendors race towards patching vulnerabilities in their software, there are two additional problems that entails by generating a patch. The intertwined relation between a patch for a vulnerability and the vulnerability itself, is that the existence of a patch reveals the existence of a vulnerability. Hence, any software that has semantic similarities with the original software where the vulnerability was identified, will be exposed to the risk of having a similar vulnerability that is un-patched.

- Any system that is dependent of an older version of the software which the vulnerability was fixed, will trigger a window of exposure starting from the patch time of the latest version of the software. A recent study find there exists thousands of systems that depend on older versions of a software that was exposed to a vulnerability for which a fix is available only in the latest version [28].
- Due to semantic similarities there may exist similar software security vulnerabilities occurring in different software systems, which are known as `recurring vulnerabilities` [106]. Hence, some software will be at the risk of been unpatched to a known vulnerability that was disclosed as a result of a patch for a semantically similar software.

The problem we address in this thesis is the problem of generating security patches, that minimizes the impact of a disclosure of a vulnerability. Since, the patch itself is a form of disclosure of a vulnerability we also need to be able to adapt a security patch generated for the identified vulnerability to all other semantically similar programs that may exhibit the potential for exploitation of a similar vulnerability. In this work, we aim to improve the capabilities of automated program repair to alleviate the problem of generating security patches. Such patches provide protection against exploitation of identified vulnerabilities. As discussed in previous sections, fixing a software security vulnerability itself is a time-consuming and difficult task. In addition, generating security patches for all variants of the vulnerability requires significant time and effort. This is particularly challenging for enterprise software vendors that consume thousands of Free and Open-Source Software(FOSS) components and offer more than a decade of support and security fixes for their applications.

1.3 Challenges

First challenge for the problem of automatically patching vulnerability is to efficiently and quickly generate patches in a time-frame less than 24 hours. Program repair techniques should be able to explore a large search-space efficiently, and generate high-quality patches in a fixed amount of time. Given the potential usage of automated program repair to alleviate the burden of generating security patches from software developers, improving the efficiency of existing repair technology is crucial.

Second challenge is the trustworthiness of auto-generated patches perceived by software developers. Albeit the ability to efficiently generate a patch, if the automated repair system cannot meet the trust requirements deemed by the software developer, the generated patch will not be applied. One of the difficulty in providing necessary guarantees for the fix, is due to the inherent problem of patch overfitting, where the patched program fails for test cases not provided by the given test-suite [69, 110]. This is an undesirable property in the context of fixing software security vulnerabilities.

The third challenge is to generalize the patch and adopt the fix to be applicable to older versions of the software, which most likely exhibits a similar variant of the identified error. The sheer complexity of the patches, the diversity of the transformations involved, and the absence of test cases (provided test-case may not be applicable to older versions) as specification pose additional challenges to port a patch across different versions of the same software.

The fourth challenge is to repair other similar security vulnerabilities in a software ecosystem, the patch should be applied in other similar software which may exhibit potential for exploitation due to semantic similarities. The challenge in incorporating patches from different sources is to be able to adapt the code modifications involved. Often, shared libraries are customized with new features, different data structures or rewriting previous implementations to match the integrated environment. Hence, directly applying a general patch is not trivial and sometimes difficult.

1.4 Thesis Overview

To achieve the goal of alleviating the problem of automatically generating security patches, we first investigate the impeding challenges in the adoption of program repair by software practitioners to fix vulnerabilities/bugs in real-world applications, and address these challenges by proposing a series of cohesive techniques that address the above tightly connected challenges.

- **Trust in Program Repair** Trust in automatically generated patches is necessary for achieving our goal of using program-repair to automatically generate security patches. Towards this goal, we survey more than 100 software practitioners to understand the artifacts and setups needed to enhance trust in automatically generated patches. Based on the feedback from the survey on developer preferences, we quantitatively evaluate existing test-suite based program repair tools. We find that they cannot produce high-quality patches within a top-10 ranking and an acceptable time period of 1 hour. The developer feedback from our qualitative study and the observations from our quantitative examination of existing repair tools point to actionable insights to drive program repair towards achieving trust in software developers.

- **Compilation-Free Program Repair** Existing program repair techniques, such as generate & validate techniques suffer from some practical limitations. Specifically, the validation step in test-based G&V requires that the candidate patch be applied to the program and recompiled, before tested against the test suite. However, recompilation can be time consuming, especially if the G&V repair tool needs to validate thousands of potential patch candidates. We propose to accelerate G&V-based repair by essentially removing the compilation step, thereby achieving the goal of Compilation Free program Repair (CFR). The basic idea is to use a combination of binary rewriting and patch interpretation to directly validate candidate patches "on-the-fly" without the need for program recompilation. This would allow for navigation of significantly larger search spaces for program repair within a time limit which is crucial to repair security vulnerabilities.

- **Concolic Program Repair** Existing program repair techniques modify a buggy program such that it passes given tests. Such repair techniques do not discriminate between correct patches and patches that overfit the available tests. We propose an integrated approach to provide guarantees of software security specification(s) via systematic co-exploration of the patch space and input space. We leverage concolic path exploration to systematically traverse the input space (and generate inputs), while ruling out significant parts of the patch space. Our technique provides guarantees for generating high-quality patches with respect to a user-provided program specification, in addition we generate additional test-cases to increase the confidence for the correctness of generated patches.

- **Automated Patch Backporting** We propose a patch backporting technique that can automatically transfer patches from one version of a software into other older versions. Our approach aims to generalize and backport security patches across different versions of the same software. We first synthesize a partial transformation rule based on a single patch. This rule can then be generalized by analysing the alignment between two different versions of the same program, exploiting the syntactic similarity between the two programs. The generalized rule is then applied to the target version to produce a backported patch. Compared to existing techniques, our approach improves both the precision and recall in backporting patches. The proposed method also helps in reducing the exposure to known security vulnerabilities in older versions of the same software.

- **Automated Patch Transplantation** A patch generated for an error in a known software program is automatically adapted and inserted into a “similar” target program. This work aims to port security patches across different software systems, to improve protection of the overall software ecosystem. We propose and implement a workflow for transplanting patches, across syntactically different yet semantically similar programs. Our approach centers on identifying patch insertion points, as well as namespace translation across programs via symbolic execution. The proposed method help mitigate similar vulnerabilities in different software systems known as recurring vulnerabilities.

Research Scope

In this thesis, we restrict our work to fix identified software security vulnerabilities. Our work does not investigate zero-day vulnerabilities which are vulnerabilities that exist but not disclosed nor discovered. Zero-day vulnerabilities pose a higher risk to users because malicious users race to exploit these vulnerabilities and the vulnerable systems are exposed until a patch is issued by the vendor. A patch to a zero-day vulnerability would not be generated until its discovery or disclosure. Except for zero-day vulnerabilities, our proposed solutions should help minimize the impact of already identified vulnerabilities by quickly generating patches to mitigate similar variants of the identified vulnerability.

1.5 Contributions and Impact

1.5.1 Conceptual Contributions

Despite the benefits, previous syntactic program repair techniques suffered from the problem of overfitting patches, since the specification was inferred from tests which only captured the correctness criteria of "passing the tests". Semantic repair techniques showed promise to infer specification representing the meaning of the defect compared to just "passing the tests", that provided additional correctness guarantees. Early work on semantic program repair extracted specification that effectively captures the semantics of the whole program [100], which inadvertently introduced scalability issues when applied to large-scale programs. More recent work [92] addressed the scalability issues by extracting a concise semantic-signature that scales constraint-based repair to large-scale programs. We introduce a novel technique "concolic program repair" which takes a balanced approach by taking a viewpoint of "*gradual correctness*" which starts from the concise semantics presented by [92] and gradually improve the specification to the extent of inferring the specification for the whole program. Exploring this viewpoint with an anytime algorithm allows the user to decide on the level of correctness enforced as repair constraints, starting from the concise correctness provided by the tests to the extent of whole program correctness. Using an anytime algorithm, we can return a partially correct set of patches, whose quality depends on the amount of verification which it was able to perform. Hence, we show that program repair problem can be formulated as an anytime algorithm which provides flexibility in time and resources, while making sure the output is better quality in return for turn-around time. This notion of "*gradual correctness*" can also be meaningful for program synthesis and transplantation whereby producing high quality automatically generated code.

Another limitation in semantic program repair is the path explosion problem inherited from classical symbolic execution, that might limit its effectiveness. In recent work this limitation was addressed by symbolic execution with existential second-order constraints which considered the search space into account during path exploration [90]. Raising the order of path constraints, one can effectively check for path infeasibility in the context of considered language of interpretations (for program repair, the patch space), thus allowing to efficiently prune irrelevant paths. This approach allows to reason about the input-space from the context of the patch-space. We take a step further to simultaneously reason about the patch-space from the context of the input-space, as well. The systematic co-exploration of the patch-space and input-space can be realized using symbolic execution by computing the path constraints for the exploration in terms of input variables and program variables. This allows us to use a light-weight symbolic execution such as concolic execution to drive the exploration, while providing the semantic reasoning to check for patch infeasibility with respect to a given program-specification. Such a co-exploration enables systematic traversal of a large-search space efficiently. One could potentially replace symbolic execution with other automated test generation techniques in our method, such as recent systematic versions of greybox fuzzing [13]. Conceptually, we present the idea of systematic co-exploration of the input space and patch space which leads to less over-fitting patches, over time.

Once a “trusted” correct patch is generated or manually written by the developer, the next step would be to adapt the patch to older versions of the same program. Early work on backporting formulated the problem as a program transformation problem where the goal was to find the most general transformation that is applicable to a large input-space. Program transformation techniques attempt to capture the most-generalized transformation by abstracting the context [93]. If the transformation is over-generalized it will lead to higher number of false-positives and it under-generalized it will lead to a higher number of false-negatives. Therefore, we once more take a balanced viewpoint that the transformation rule should strike a balance such that the context is not over-generalized and the transformation is not over-specific. Hence, we formulate the backporting problem as a program synthesis problem where the goal is to synthesise the correct transformation rule for the target context. In comparison to existing program transformation techniques which search for a single transformation rule that can be applied for all input, the goal should be to generate for each version the transformation rule that match the target context. Re-purposing program synthesis technique for program transformation to backport patches, shows significant improvement in our experiment results.

1.5.2 Technological Contributions

In summary, the core technological contributions made by this thesis are:

- A novel technique to systematically co-explore the input space and program space, which can incorporate a user provided program-specification to guide program repair for high-quality patches. Our “concolic program repair” technique can efficiently explore a large-search space, producing additional test-cases for validation and ranking high-quality patches while gradually improving the overall correctness of the generated patches.

- A novel transformation rule synthesis algorithm that can produce properly generalized transformation rules to automate the patch backporting process. Proposed patch backporting technique can synthesise transformation-rules from a single example, which allows synthesis of different transformations for different programs.

- The PatchWeave transplantation technique, performs concolic execution driven code transplantation which can automatically identify the insertion point for the transplantation based on the semantic similarities of any two programs that implement a higher-level abstraction such as a standard or a protocol.

Based on the above proposed concepts, we have developed and open-sourced three tools, CPR, FixMorph and PatchWeave.

- CPR, which can incorporate user provided program-specification to filter over-fitting patches, is open-sourced at https://github.com/rshariffdeen/cpr
- FixMorph, which can automatically backport patches for the Linux kernel project, is open-sourced at https://github.com/rshariffdeen/fixmorph
- PatchWeave, which can automatically extract and transplant patches across semantically similar programs is, open-sourced at https://github.com/rshariffdeen/patchweave

1.5.3 Empirical Contributions

We believe in the Open Science approach, openness in scientific work is key to fostering progress via transparency and availability of all outputs produced at each investigative step. Transparency and availability of research outputs allow the research community to better replicate and reproduce the findings in our quantitative studies and recover information from our qualitative studies. Open science builds the core for excellence in evidence-based research. Adhering to the principles of Open Science, we explicitly committed ourselves to foster openness to our research outcomes and made our artifacts available:

- All our findings from of our study on automated program repair and the quantitative and evaluation of the program repair tools are made available via https://doi.org/10.5281/zenodo.5376904. Inclusive of reproducible codes for the survey conducted with software practitioners and setup scripts to perform the quantitative experiments on existing program-repair tools. Our submitted artifact was evaluated and awarded all three reproducible badges by the Artifact Evaluation Committee, at the 44th IEEE/ACM International Conference on Software Engineering (ICSE'22).

- Artifacts on all our experiments on concolic program repair are made available via https://cpr-tool.github.io. Our submitted artifact was evaluated and awarded all three reproducible badges by the Artifact Evaluation Committee, at the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI'21).

- Data on the empirical study of the Linux backporting efforts and all our experiments on automated patch backporting inclusive of the comparison studies are made available via https://fixmorph.github.io. Our submitted artifact was evaluated and awarded the most distinguished artifact award by the Artifact Evaluation Committee, at the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'21).

- Artifacts on all our experiments on automated patch transplantation are made available via https://patchweave.github.io

Our contributions impact the current state of practice of vulnerability fixing by software developers and the research community in developing techniques to preserve security of software. Our technical contributions make an impact by assisting developers to repair software security vulnerabilities efficiently and in a timely manner. Specifically, the developed concepts and techniques enables developers to automatically and efficiently fix a wider range of security vulnerabilities in real-world software. It can also provide additional guarantees that were previously missing in existing program repair techniques, that the auto-generated patches are correct. This greatly impacts the productivity of the developers, helps mitigate software vulnerabilities and increase the security of software systems. Our empirical contributions make an impact by assisting researchers in the software engineering community to evaluate/investigate further research directions using our findings and reproducible artifacts.

1.6 Thesis Outline

This thesis first introduces the background knowledge necessary for this work in Chapter 2, we provide an overview in symbolic execution, program synthesis, program repair and software transplantation. Next, we present our study on developer's perspective on automated program repair and identify the impeding challenges to incorporate automatically-generated patches to fix vulnerabilities in real-world applications in Chapter 3. Chapter 4 introduces the concept of compilation free repair as a solution to improve the efficacy of generate & validate repair techniques by removing the compiler-in-the-loop from the repair process, enabling significant speedup for the overall repair process. In Chapter 5 we introduce a novel repair technique, 'concolic program repair' which can incorporate a user-provided program-specification to guide program repair, which can address limitations in existing program-repair to fix security-vulnerabilities. Chapter 6 presents an empirical study on the backporting problem in Linux kernel, which has a significant impact on software system security and provides a solution for the backporting problem inspired from patch-synthesis technique. Chapter 7 introduces to the problem of automated patch transplantation (which addresses a specific class of security vulnerabilities known as recurring vulnerabilities) and evaluate our proposed solution based on a combination of concolic execution and symbolic reasoning. Chapter 8 presents the related work and Chapter 9 concludes this thesis by summarizing all the presented techniques and improvement, and we discuss potential future research directions based on our current contributions.

Chapter 2

Background

This chapter introduces the necessary background knowledge in the areas of software security, symbolic execution, program synthesis, program repair and software transplantation.

2.1 Software Security Vulnerabilities

A software system is a collection of software programs and/or dependent library modules. A bug in a software refers to an observable error that produces an incorrect result or an unintended behavior. A software security vulnerability is a software bug that can be taken advantage of to disrupt the intended behavior of the software system. Such software vulnerability can aid malicious users to misuse a software application and circumvent deployed security measures for the system.

A **zero-day vulnerability** is a security vulnerability in a software system or hardware device that has been identified but not yet disclosed, thus not yet patched. Usually discovered by malicious users before the vendor of the software systems has become aware of it. An attacker who identifies a zero-day vulnerability will retain its viability with reasonable confidence for higher probability of success for a long period in the order of years. Large-scale exploitation of such zero-day vulnerabilities may result in detection and subsequent remediation, hence an attacker uses such vulnerabilities for targeted attacks which persist unnoticed.

An exploit is a software code that takes advantage of an existing software security vulnerability. It is usually generated by security researchers as a proof-of-concept threat for further investigations or to aid remediation. Malicious users generate exploits to further their operations. A **zero-day exploit** is a certain type of exploit that targets zero-day vulnerabilities to take advantage of the non-existence of its discovery. The act of using zero-day exploits to cause damage to or steal data from a software system affected by zero-day vulnerability is known as a **zero-day attack**. There is almost zero defense against a zero-day attack due to the nature of being unknown. Since the vulnerability remains unknowns the software affected cannot be patched nor detected using anti-virus products that uses signature-based scanning.

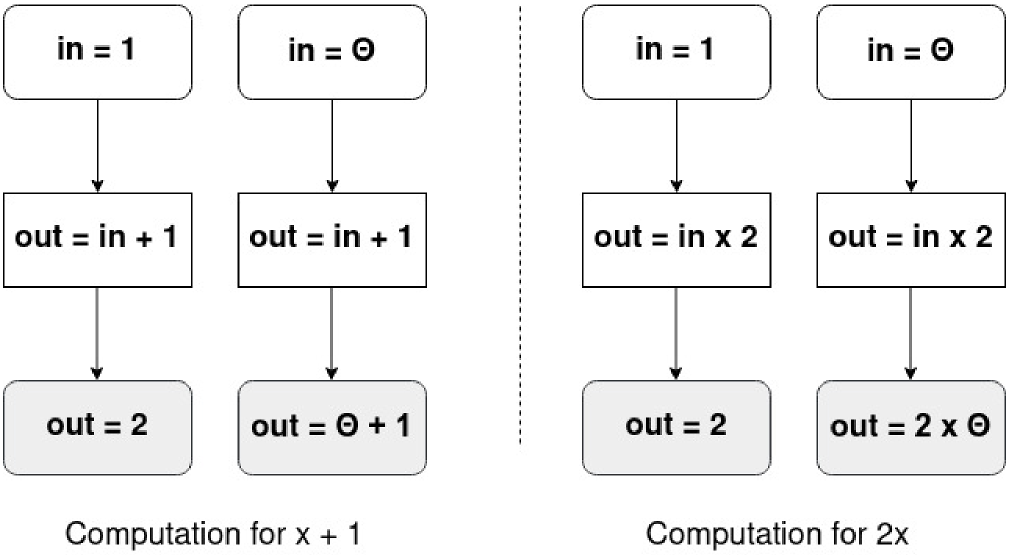
Although zero-day attacks pose a significant threat to the security of the software system, there exist another class of vulnerabilities that poses a much larger concern due to its readily available active exploits and widely available knowledge from public disclosure documents. **N-day attacks** use already disclosed security vulnerabilities that may or may not have a security patch available. The `N` in N-day attack signifies the number of days between the time of disclosure of the vulnerability and the time of the attack. The goal of malicious attackers is to capitalize on this time window while the goal of the software vendors, distributors is to patch such systems as quickly as possible.

2.2 Symbolic Execution

Symbolic execution, introduced in the mid '70s to test whether certain properties can be violated by a piece of software [15, 50, 60, 61] has recently emerged as an effective technique for generating high-coverage test suites and for finding deep errors in complex software programs [16, 42], as a result of significant advances in constraint satisfiability solvers. Typically used in testing to explore program paths, generate a set of concrete witnesses (input values) exercising a given path. Contrast to concrete execution where a program is run on a specific input and a single control flow path is explored, symbolic execution is performed by a symbolic execution engine, which maintains for each explored control flow path: (i) a first-order boolean formula that describes the conditions satisfied by the branches taken along that path, and (ii) a symbolic memory store that maps variables to symbolic expressions or values.

Figure 2.1 illustrates the difference between symbolic execution and concrete execution for the two computations x+1 and 2x. Consider the output of the two concrete executions, which result in the concrete value 2. Analysing the output of a

Figure 2.1: Comparison of symbolic execution and concrete execution.



On the left hand side the computation for (x + 1) with the leftmost diagram depicting the concrete execution and alongside with the symbolic execution. Similarly, on the right hand side the computation for (2x) with the rightmost diagram depicting the symbolic execution alongside with the concrete execution to its left.

concrete execution does not provide us any insight on the semantic meaning of the output, and cannot differentiate the two computations are different based on the concrete output alone, unless multiple such output values are collected. In contrast symbolic execution allows us to not only differentiate the two computations based on a single execution, but also provide semantic meaning for the two outputs. The semantic meaning, helps us to perform program analysis in an efficient and scalable manner, which we will explore in the next section of program repair. In the context of software security, it has been widely used to provide guarantees for the non-existence of division by zero, NULL pointer dereferences and other security/software properties. While in general there is no automated way to decide some properties (e.g., the target of an indirect jump), heuristics and approximate analyses can prove useful in practice in a variety of settings, including mission-critical and security applications [7].

There are two artifacts of symbolic execution that are of use for program analysis and program repair. **Symbolic Expression** is a formula derived from the operations over symbolic inputs. **Symbolic Path Condition** is a first-order boolean formula that describes the conditions satisfied by the branches taken along that path.

Figure 2.2: Example code for Symbolic Execution

```
 1    input in=5;
 2    a=0; b=0;
 3    if (in > 3){
 4      a = in * 2;
 5      b = in + 2;
 6      b = b + 2;
 7    }
 8
 9    if (a > b){
10      return error;
11    }
```

Figure 2.2 shows a sample C code snippet for a symbolic execution in which we mark program variable `in` as symbolic (denoted by θ), and Table 2.1 records the symbolic store and symbolic path condition along each line. The symbolic store keeps track of each variable observed along a program execution with the symbolic expression for the variable. The path condition reflects the branch conditions that was satisfied along a given execution path.

Assuming line number 4 of Figure 2.2 is executed, it should satisfy the first branch condition (`in` > 3), since we mark the variable `in` as symbolic θ, our symbolic path condition will be updated as ($\theta > 3$). Similarly, the value of program variable `a` will be updated to the symbolic expression (a, 2θ) since the new value of `a` is computed using a symbolic variable. The symbolic store and the path condition will be updated as the execution continues. Executing line number 10 in Figure 2.2 depends on the satisfiability of the branch condition at line number 9. Using SMT solvers [142] we can compute such satisfiabilities which allows us to perform advance program analysis alone a single symbolic execution. Depending on the specific analysis we perform using symbolic execution, various kinds of semantic reasoning and/or path exploration can be performed for the given test code. For this example run, if the satisfiability holds true, the program will return an error which can be then computed as a constraint to generate an input for the observed error.

There are several types of symbolic execution of interest:

Table 2.1: Illustration of symbolic execution

Line	Symbolic Store	Path Condition
1	$\{\}$	(true)
2	$\{$(a, 0), (b, 0)$\}$	(true)
3	$\{$(a, 0), (b, 0)$\}$	$(\theta > 3)$
4	$\{$(a, 2θ), (b, 0)$\}$	$(\theta > 3)$
5	$\{$(a, 2θ), (b, θ+2)$\}$	$(\theta > 3)$
6	$\{$(a, 2θ), (b, θ+4)$\}$	$(\theta > 3)$
7	$\{$(a, 2θ), (b, θ+4)$\}$	$(\theta > 3)$
9	$\{$(a, 2θ), (b, θ+4)$\}$	$(\theta > 3) \wedge (2\theta > (\theta + 4))$

- **Classical Symbolic Execution:** a symbolic execution of a program that can generate(in theory) all possible control flow paths that the program could take during its concrete executions on specific inputs. While modeling all possible runs allows for very interesting analyses, it is typically infeasible in practice, especially on real-world software.

- **Concolic Execution:** the term concolic is a portmanteau of the words “concrete” and “symbolic”. This technique makes use of concrete input to drive the symbolic execution, compared to classical execution where the inputs are purely symbolic. There are multiple approaches using concolic execution such as dynamic symbolic execution as used in DART [42] and SAGE [43], selective symbolic execution as used in S2E [26] and MAYHEM [18].

- **Symbolic Backward Execution:** a variant of symbolic execution in which the exploration proceeds from a target point to an entry point of a program. The analysis is thus performed in the reverse direction than in canonical (forward) symbolic execution. The main purpose of this approach is typically to identify a test input instance that can trigger the execution of a specific line of code (e.g., an assert or throw statement) [29, 22, 84].

2.3 Program Synthesis

Synthesis is a combination of components or elements to form a connected whole, similarly program synthesis is a methodology that can automatically construct programs from some predefined set of components (i.e. synthesis language), while adhering to a given user specification (requirements). Specification is defined as `input`↦`output` pairs which describes the intended behavior of the program. Specifically, for a given input domain $\mathbb{I}$ and output domain $\mathbb{O}$, the synthesis technique should take as input a list of pairs $\{i_0 \mapsto o_0, ..., i_n \mapsto o_n\}$ and generate a program $P : \mathbb{I} \to \mathbb{O}$ such that $P(i_k) = o_k$ for all $0 \leq$ k $\leq$ n.

2.3.1 Component-based Program Synthesis

Recent work formalized program synthesis from the logical point of view, and considered as a second-order constraint solving problem [90]. Initially Jha et al. [53] proposed an approach to encode the search space using linear integer arithmetic constraints. Mechtaev et al. [90] represented the search-space as circuits built from user-provided components such as addition("+"), subtraction("-") etc. where connection between components are captured using integer location variables. Thus, a program can be reconstructed by a valuation of location variables which can be achieved using an SMT solver.

2.3.2 Code Transformation

Program synthesis technique can be used to automatically transform code using its Abstract Syntax Tree (AST) representation. ReFazer [113] is one such technique known for code transformations. It uses its own domain-specific language(DSL) where a transformation is formulated as a function pair (`guard`, `transformer`). We denote the set of typed ASTs as $\mathbb{T}$. A transformation rule $R : \mathbb{T} \to \mathbb{T}$ formulates how to transform a $\mathbb{T}$ to another $\mathbb{T}$. Rule R can be represented as a pair (`guard`, `transformer`) similar to [113, 95] defined as follows:

- `guard`: $\mathbb{T} \to$ `Boolean`: `guard` is a conjunction of predicates over AST nodes. Basically, a `guard` tests the type, code and other attributes of an AST node and returns a Boolean value representing whether the node satisfies its predicate or not;

- `transformer`: $\mathbb{T} \to \mathbb{T}$: `transformer` takes an input $\mathbb{T}$ and constructs another $\mathbb{T}$. It is built from two underlying operations: (1) `select`: returns an existing node from input $\mathbb{T}$ satisfying a given *guard*, and (2) `construct`: returns a new node constructed from a specific node kind, attributes, and children.

Essentially, the rule `guard` determines which AST sub-node should be transformed, and the `transformer` determines how the sub-node should be transformed. Thus, for $t \in \mathbb{T}$, we have $R(t) = \texttt{transformer}(t)$ when $\texttt{guard}(t)$ is `true`, otherwise, $R(t)$ is $\perp$.

2.4 Program Repair

Program Repair is the problem of minimally modifying a program with the goal of fixing an identified error or remedying an unintended behavior. Automated Program Repair requires providing a correctness criteria that may include both functional and non-functional requirements. Existing program repair techniques derive such specification from a test-suite and possibly other quality metrics that can be used as a fitness function to evaluate the best modification. Several repair techniques have been studied in the literature with the goal of finding the correct program that pass the given test-suite thus fixing an identified/observable bug.

2.4.1 Search-Based Repair

Program Repair problem can be formulated as a search problem where the goal is to search for the correct patch (i.e. program modification) in a large-space consisting of different patches, that can meet the specification defined by the test-suite. The first step is to generate a search space (i.e. patch space) `S` [80] and then search among `S` to find the patch `P` that satisfies the correctness requirements. Different techniques have been proposed to define the search-space based on a set of predefined repair operators such as mutation [68] and templates derived from human-written patches [79, 78]. Once the search-space is generated, the algorithm would then search for the correct patch using various search heuristics. The process terminates once a program is found to pass the correctness criteria, the found patch is known as a ‘plausible patch‘. This resulted in generating patches that are overfitting the given test-suite and fails on test-cases outside of the provided test-suite. Several approaches have been studied to alleviate the overfitting problem using anti-patterns [138] and additional test-generation [39]. Furthermore, traditional repair techniques enumerates the search-space one at a time and evaluate each patch against the provided test-suite, which is inefficient and time-consuming thus does not scale for programs with a large search-space. Recent studies have shown to improve the space exploration using test-equivalence analysis [89, 58] which can optimize the search process.

2.4.2 Semantic-Based Repair

Program Repair problem can also be formulated as a constraint-solving problem which initially construct constraints that should be satisfied to repair a bug, and then use program synthesis (i.e. component-based synthesis [53]) to synthesise a patch that satisfies the generated repair constraints. The constraints for the correctness criteria is derived from the provided test-suite, hence the initial step is to extract the specification as repair-constraints from the provided test-suite, usually achieved with symbolic execution. For each identified suspicious location in the program, traditional semantic-repair techniques [100, 161, 91] would inject symbolic variables and derive repair constraints by running against the test-suite. Once the specification is inferred, it will be used to synthesize a replacements for the suspicious location that would enable the program to pass the initially failing test cases. Since the specification is derived from the provided test-suite, the repair generated also inherits the problem of overfitting. Furthermore, since the use of specification inference is dependent on symbolic execution [100], it suffers from the path explosion problem in symbolic execution thus limits its effectiveness. Recent work on semantic-repair takes a different approach of specification inference that extracts concise synthesis specification instead of whole program [92], which also allowed to repair complex multi-line changes. The path explosion problem in symbolic execution has also been addressed with existential second-order constraints [90] that considers the search-space during path exploration.

2.4.3 Learning-Based Repair

Program Repair problem can also be formulated as a code transformation problem, where the patch generation is considered as a transformation of the Abstract Syntax Tree(AST) representation of the program. These techniques [77, 6, 10] mine for patterns from large code-bases to learn different kinds of transformations that can fix reported bugs in open-source software repositories. Since the approach learns repair strategies by mining existing patches, they do not require a set of predefined transformation operators. More recent work on neural machine translation for program repair also known as "neural program repair" is based on encoder-decoder architecture optimized with cross-entropy loss function that can translate the buggy code into a correct code [165, 167, 147, 83, 57, 48, 25, 19, 20]. This line of work represent the program as code edits for the AST, where a supervised learning model learns from a large training set of previous developer commits. The loss function is computed as the difference between the generated tokens and the human-written patch tokens in a strict pairwise matching, which can be used to adjust the weights of the model to obtain a result that is closer to the ground truth at the token-level.

2.5 Software Transplantation

Transplantation is a technique of introducing foreign organ to an existing body, and similarly in the context of software, injecting foreign code from a different software is known as software transplantation[9, 126, 125]. In the transplantation literature a *organ* is a piece of code (a function, a test case, a validation check etc.) which will be transferred, a *donor* is a program which provides the organ that will be transplanted, and the *host/target/receiver* is the program which the organ will be injected into.

A programmer must first identify a compatible pair where the organ can be shared, i.e. code similarity, implementation, and design similarity. Once a pair of programs have been identified, the organ that needs to be transplanted should be extracted (all code associated with the feature/test/block) and must identify the implantation point (insertion point) in the target. The extraction of the code also involves identifying all semantically required code and the successful insertion of the code organ into the host requires nontrivial modifications to the organ to ensure it adds the required feature without breaking existing functionality. Translation of namespace and adaptation of data-structures would be required for the transfer.

Chapter 3

Developer's perspective on Automated Program Repair

Automated program repair is an emerging technology that seeks to automatically rectify vulnerabilities and bugs using learning, search, and semantic analysis. Trust in automatically generated patches is necessary for achieving greater adoption of program repair. Especially when fixing high-impact time-critical security vulnerabilities which could lead to potential economical and financial losses. Towards this goal, we survey more than 100 software practitioners to understand the artifacts and setups needed to enhance trust in automatically generated patches and identify impeding challenges to incorporate an automated solution to fix vulnerabilities and bugs. Based on the feedback from the survey on developer preferences, we quantitatively evaluate existing test-suite based program repair tools. In this chapter we study the developer expectation of the role of automated program repair in the software development life cycle. More specifically we study the expectations of the developers and the impeding technical challenges in practice to integrate auto-generated patches in real-world applications. This chapter starts with an overview on how automated program repair works and what correctness guarantees are provided to trust auto-generated patches. It continues with a qualitative study with software practitioners to gain insight on the expectations of program repair tools. Afterwards, a quantitative analysis of existing automated program repair tools will be presented with respect to the developer expectations. The chapter concludes with actionable insights to drive program repair research, specifically in the direction of trusted repair.

3.1 Overview

Automated program repair technologies [69] are getting increased attention. In recent times, program repair has found its way into the automated bug fixing of mobile apps in the SapFix project in Facebook [85], automated repair bots as evidenced by the Repairnator project [148], and has found certain acceptability in companies such as Bloomberg [62]. While all of these are promising, large-scale adoption of program repair where it is well integrated into our programming environments is considerably out of reach as of now. In this chapter, we reflect on the impediments towards the usage of program repair by developers. There can be many challenges towards the adoption of program repair like scalability, applicability, and developer acceptability. A lot of the research on program repair has focused on scalability to large programs and also to large search spaces [79, 92, 39, 85]. Similarly, there have been various works on generating multi-line fixes [92, 40] to cover various use cases or scenarios of program repair.

Surprisingly, there is no literature or systematic study from either academia or industry on the developer trust in program repair. In particular, what changes do we need to bring into the program repair process so that it becomes viable to have conversations on its wide-scale adoption? Part of the gulf in terms of lack of trust comes from a lack of specifications since the intended behavior of the program is not formally documented, it is hard to trust that the automatically generated patches meet this intended behavior. Overall, we seek to examine whether the developer's reluctance to use program repair may partially stem from not relying on automatically generated code. This can have profound implications because of recent developments on AI-based pair programming[1], which holds out promise for significant parts of coding in the future to be accomplished via automated code generation.

In this chapter, we specifically study the issues involved in enhancing developer trust on automatically generated patches. Towards this goal, we first settle on the research questions related to developer trust in automatically generated patches. These questions are divided into two categories (a) expectations of developers from automatic repair technologies, and (b) understanding the possible shortfall of ex-

[1] Github Copilot `https://copilot.github.com/`

isting program repair technologies with respect to developer expectations. To understand the developer expectations from program repair, we outline the following research questions.

- **RQ1** To what extent are the developers ready to accept and apply automated program repair (henceforth called APR)?
- **RQ2** Can software developers provide additional inputs that would cause higher trust in generated patches? If yes, what kind of inputs can they provide?
- **RQ3** What evidence from APR will increase developer trust in the patches produced?

For a comprehensive assessment of the research questions, we engage in both qualitative and quantitative studies. Our assessment of the questions primarily comes in three parts. To understand the developer expectations from program repair, we conduct a detailed survey (with 35 questions) among more than 100 professional software practitioners. Most of our survey respondents are developers, with a few coming from more senior roles such as architects. The survey results amount to both quantitative and qualitative inputs on the developer expectations since we curate and analyze respondents' comments on topics such as desired evidence from automated repair techniques. Based on the survey findings, we note that developers are largely open-minded in terms of trying out a small number of patches (no more than 10) from automated repair techniques, as long as these patches are produced within a reasonable time, say less than 1 hour. Furthermore, the developers are open to receiving specifications from the program repair method (amounting to evidence of patch correctness). They are also open-minded in terms of providing additional specifications to drive program repair. The most common specifications the developers are ready to give and receive are tests.

Based on the comments received from survey participants, we then conduct a quantitative comparison of certain well-known program repair tools on the widely used ManyBugs benchmarks [46]. To understand the possible deficiency of existing program repair techniques with respect to outlined developer expectations as found from the survey, we formulate the following research questions.

RQ4 Can existing APR techniques pinpoint high-quality patches in the top-ranking (e.g., among top-10) patches within a tolerable time limit (e.g., 1 hour)?

RQ5 What is the impact of additional inputs (say, fix locations and additional passing test cases) on the efficacy of APR?

We note that many of the existing papers on program repair use liberal timeout periods to generate repair, while in our experiments the timeout is strictly maintained at no more than one hour. We are also restricted to observing the first few patches, and we examine the impact of the fix localization by either providing and not providing the developer location. Based on a quantitative comparison of well-known repair tools Angelix [92], GenProg [68], Prophet [79] and Fix2Fit [39] — we conclude that the search space representation has a significant role in deriving plausible/correct patches within an acceptable time period. In other words, an abstract representation of the search space (aided by constraints that are managed efficiently or aided by program equivalence relations) is at least as critical as a smart search algorithm to navigate the patch space. We discuss how the tools can be improved to meet developer expectations, either by achieving compilation-free repair or by navigating/suggesting abstract patches with the help of simple constraints (such as interval constraints).

Last but not the least, we note that program repair can be seen as automated code generation at a micro-scale. By studying the trust issues in automated repair, we can also obtain an initial understanding of trust enhancement in automatically generated code. Which plays a vital role in adapting auto-generated patches to fix security vulnerabilities.

3.2 Specifications in Program Repair

The goal of APR is to correct buggy programs to satisfy given specifications. In this section, we review these specifications and discuss how they can impact patch quality.

Test Suites as Specification

APR techniques such as GenProg [68], Prophet [79], treat test-suite as correctness specification. The test suite usually includes a set of passing tests and at least one failing test. The repair goal is to correct the buggy program to pass all the given test suites. Although test suites are widely available, they are usually incomplete specifications that specify part of the intended program behaviors. Hence, the automatically generated patch may overfit the tests, meaning that the patched program may still fail on program inputs outside the given tests. For instance, the following is a buggy implementation that copies `n` characters from source array `src` to destination array `dest`, and returns the number of copied characters.

```
1 int lenStrncpy(char[] src, char[] dest, int n){
2     if(src == NULL || dest == NULL)
3         return 0;
4     int index = -1;
5     while (++index < n)
6         dest[index] = src[index]; // buffer overflow
7     return index;
8 }
```

Figure 3.1: Illustrative example for patch-overfitting problem

A buffer overflow happens at line 6 when the size of `src` or `dest` is less than `n`. By taking the following three tests (one of them can trigger this bug) as specification, a produced patch (++`index`<`n` ↦ ++`index`<`n` && `index`<3) can make the program pass the given tests. Obviously, the patched program is still buggy on test inputs outside the given tests.

Table 3.1: Illustrative test-suite for patch-overfitting problem

Type	src	dest	n	Output	Expected Output
Passing	SOF	COM	3	3	3
Passing	DHT	APP0	3	3	3
Failing	APP0	DQT	4	*crash	3

Constraints as Specification

Instead of relying on tests, another line of APR research, e.g., ExtractFix [40] take constraints as correctness specifications. Constraints have the potential to represent a range of inputs or even the whole input space. Driven by constraints, the goal of APR is to patch the program to satisfy the constraints. However, different from the test suite, the constraints are not always available in practice; for this reason, techniques like ANGELIX [92] and SemFix [100] take tests as specifications but extract constraints from tests. Certain existing APR techniques take as input coarse-grained constraints, such as assertions or crash-free constraints. For instance, EXTRACTFIX relies on predefined templates to infer constraints that can completely fix vulnerabilities. For the above example, according to the template for buffer overflow, the inferred constraint is `index<sizeof(src)&&index<sizeof(dest)`. Once the patched program satisfies this constraint, it is guaranteed that the buffer overflow is completely fixed. Guarantees from such fixing of overflows/crashes do not amount to full functionality correctness guarantee of the fixed program.

Code Patterns as Specification

Besides test suites and constraints, code patterns can also serve as specifications for repair systems. Specifically, given a buggy program that violates a code pattern, the repair goal is to correct the program to satisfy the rules defined by the code pattern. The code patterns can be manually defined [138], from static analyzers [146], automatically mined from large code repositories [6, 10], etc. Similar to the inferred constraints, code patterns cannot ensure functionality correctness.

3.3 Developer Survey

Since constructing formal program specifications is notoriously difficult, the specifications used by APR tools cannot ensure patch correctness. Unreliable overfitting patches cause developers to lose trust in APR tools. It motivates us to enquire/-survey developers on how APR can be enhanced to gain their trust.

3.3.1 Survey Methodology

We designed and conducted a survey with software practitioners, specifically to answer the first three research questions (RQ1-3). In June 2021, we distributed a questionnaire to understand how developers envision the usage of automated program repair and what can be provided to increase trust in automatically generated patches. Note that we followed our institutional guidelines and received approval from the Institutional Review Board (IRB) of our organization, prior to administering the survey.

Survey Instrument

We asked in total 35 questions about how trustworthy APR can be deployed in practice. Our questions are structured into six categories:

C1 *Usage of APR* (RQ1): whether and how developers would engage with APR.

C2 *Availability of inputs/specifications* (RQ2): what kind of input artifacts developers can provide for APR techniques.

C3 *Impact on trust* (RQ2): how additional input artifacts would impact the trust in generated patches.

C4 *Explanations* (RQ3): what kind of evidence/explanation developers expect for auto-generated patches.

C5 Usage of APR side-products (RQ3): what side-products of APR are useful for the developers say for manual bug-fixing.

C6 *Background*: the role and experience of the participants in the software development process.

C1 will provide insights for RQ1, C2 and C3 for RQ2, and C4 and C5 for RQ3. The questions were a combination of open-ended questions like *"How would you like to engage with an APR tool?"* and close-ended questions like *"Would it increase your trust in auto-generated patches if additional artifacts such as tests/assertions are used during patching?"* with Multiple Choice or a 5-point Likert scale. The questionnaire itself was created and deployed with Microsoft Forms. A complete list of our questions can be found in Table 3.2 and in our replication package.

Participants

We distributed the survey via two channels: (1) Amazon MTurk, and (2) personalized email invitations to contacts from global-wide companies. As incentives, we offered each participant on MTurk 10 USD as compensation, while for each other

Table 3.2: List of questions from the developer survey

Category	**Question**	**Type**
C1 Usage of APR	Q1.1 Are you willing to review patches that are submitted by APR techniques?	5-Point Likert Scale
	Q1.2 How many auto-generated patches would you be willing to review before losing trust/interest in the technique?	Selection + Other…
	Q1.3 How much time would you be giving to any APR technique to produce results?	Selection + Other…
	Q1.4 How much time do you spend on average to fix a bug?	Selection + Other…
	Q1.5 Do you trust a patch that has been adopted from another location/application, where a similar patch was already accepted by other developers?	5-Point Likert Scale
	Q1.6 Would it increase your confidence in automatically generated patches if some kind of additional input (e.g., user-provided test cases) were considered?	5-Point Likert Scale
	Q1.7 Besides some additional input that is taken into account, what other mechanism do you see to increase the trust in auto-generated patches?	Open-Ended
C2 Availability of Inputs	Q2.1 Can you provide additional test cases (i.e., inputs and expected outputs) relevant for the reported bug?	5-Point Likert Scale
	Q2.2 Can you provide additional assertions as program instrumentation about the correct behavior?	5-Point Likert Scale
	Q2.3 Can you provide a specification for the correct behavior as logical constraint?	5-Point Likert Scale
	Q2.4 Would you be fine with classifying auto-generated input/output pairs as incorrect or correct behavior?	5-Point Likert Scale
	Q2.5 How many of such queries would you answer?	Selection + Other…
	Q2.6 For how long would you be willing to answer such queries?	Selection + Other…
	Q2.7 What other type of input (e.g., specification or artifact) can you provide that might help to generate patches?	Open-Ended
	Q2.8 Please describe how you would like to engage with an APR tool. For example shortly describe the dialogue between you (as user of the APR tool) and the APR tool. Which input would you pass to the APR tool? What do you expect from the APR tool?	Open-Ended
C3 Impact on trust	Q3.1 Would it increase your trust in auto-generated patches if additional artifacts such as tests/assertions are used during patching?	5-Point Likert Scale
	Q3.2 Which of the following additional artifacts will increase your trust?	Multiple Choice
	Q3.3 What are other additional artifacts that will increase your trust?	Open-Ended
C4 Explanations for generated patches	Q4.1 Would it increase your trust when the APR technique shows you the code coverage achieved by the executed test cases that are used to construct the repair?	5-Point Likert Scale
	Q4.2 Would it increase your trust when the APR technique presents the ratio of input space that has been successfully tested by the inputs used to drive the repair?	5-Point Likert Scale
	Q4.3 What other type of evidence or explanation would you like to come with the patches, so that you can select an automatically generated patch candidate with confidence?	Open-Ended
C5 Usage of APR side-products	Q5.1 Which of the following information (i.e., potential side-products of APR) would be helpful to validate the patch?	Multiple Choice
	Q5.2 What other information (i.e., potential side-products of APR) would be helpful to validate the patch?	Open-Ended
	Q5.3 Which of the following information (i.e., potential side-products of APR) would help you to fix the problem yourself (without using generated patches)?	Multiple Choice
	Q5.4 What other information (i.e., potential side-products of APR) would help you to fix the problem yourself (without using generated patches)?	Open-Ended
C6 Background	Q6.1 What is your (main) role in the software development process?	Selection + Other…
	Q6.2 How long have you worked in software development?	Selection
	Q6.3 How long have you worked in your current role?	Selection
	Q6.4 How would you characterize the organization where you are employed for software development related activities?	Selection + Other…
	Q6.5 What is your highest education degree?	Selection + Other…
	Q6.6 What is your primary programming language?	Selection + Other…
	Q6.7 What is your secondary programming language?	Selection + Other…
	Q6.8 How familiar are you with Automated Program Repair?	5-Point Likert Scale
	Q6.9 Are you applying any Automated Program Repair technique at work?	Yes/No
	Q6.10 Which Automated Program Repair technique are you applying at work?	Open-Ended

participant, we donated 2 USD to a COVID-19 charity fund. We received 134 responses from MTurk. To filter low-quality and non-genuine responses, we followed the known principles [33] and used quality-control questions. In particular, we asked the participants to *describe* their role in software development and *name* their main activity. In combination with the other open-ended questions, we have been able to quickly identify non-genuine answers. After this manual post-processing, we ended up with 34 *valid* responses from MTurk. From our company contacts, we received 81 responses, from which all have been genuine answers. From these in combination 115 valid responses, we selected **103** *relevant* responses, which excluded responses from participants who classified themselves as Project Manager, Product Owner, Data Scientist, or Researcher. Our goal was to include answers from software practitioners that have hands-on experience in software development. Figure 3.2 and 3.3 show the roles and experiences for the final subset of 103 participants.

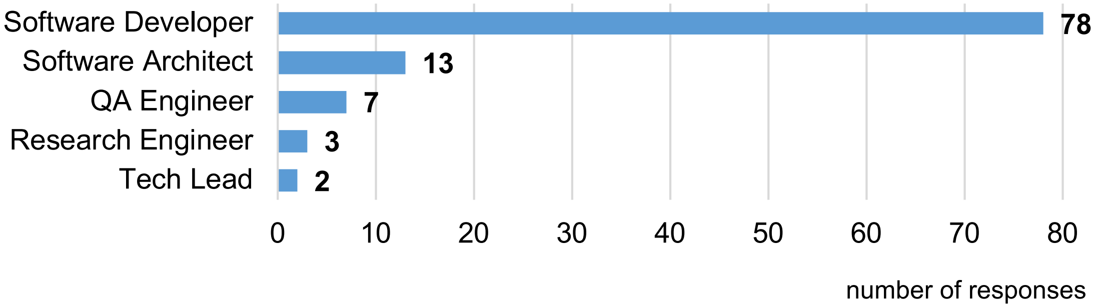


Figure 3.2: Responses for Q6.1 *What is your (main) role in the software development process?*

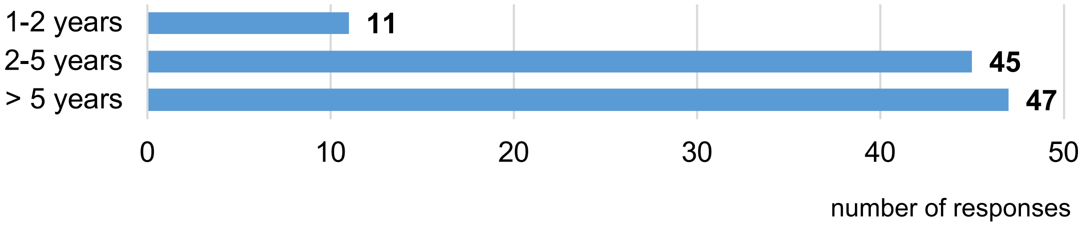


Figure 3.3: Responses for Q6.2 *How long have you worked in software development?*

Analysis

For the questions with a 5-point Likert scale, we analyzed the distribution of negative (1 and 2), neutral (3), and positive (4 and 5) responses. For the Multiple

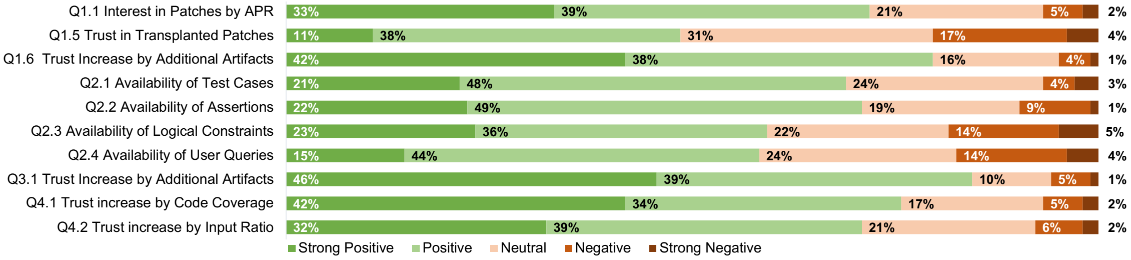


Figure 3.4: Results for the questions with the 5-point Likert Scale (103 responses).

Choice questions, we analyzed which choices were selected most, while the open-ended "Other" choices were analyzed and mapped to the existing choices or treated as new ones if necessary. For all open-ended questions, we performed a qualitative content analysis coding [117] to summarize the themes and opinions. The first iteration of the analysis and coding was done by one author, followed by the review of the other authors. In the following sections, we will discuss the most mentioned responses, and indicate in the brackets behind the responses how often the topics are mentioned among the 103 participants. All data and codes are included in our replication package.

3.3.2 Survey Results

3.3.2.1 Developer engagement with APR

In this section, we discuss the responses for the questions in category C1 and question Q2.8, which was explicitly exploring how the participants want to *engage* with an APR tool. First of all, a strong majority (72% of the responses) indicate that the participants are willing to review auto-generated patches (see Q1.1 in Figure 3.4). It generally confirms the efforts in the APR community to develop such techniques. Only 7% of the participants are reluctant to apply APR techniques in their work. As shown in Figure 3.5, we note that 72% of the participants want to review only up to 5 patches, while only 22% would review up to 10 patches. Furthermore, 6% mention that it would depend on the specific scenario. At the same time, the participants expect relatively quick results: 63% would not wait longer than one hour, of which the majority (72% of them) prefer to not even wait longer than 30 minutes. The expected time certainly depends on the concrete deployment, e.g., repair can also be deployed along a nightly Continuous Integration (CI) pipeline,

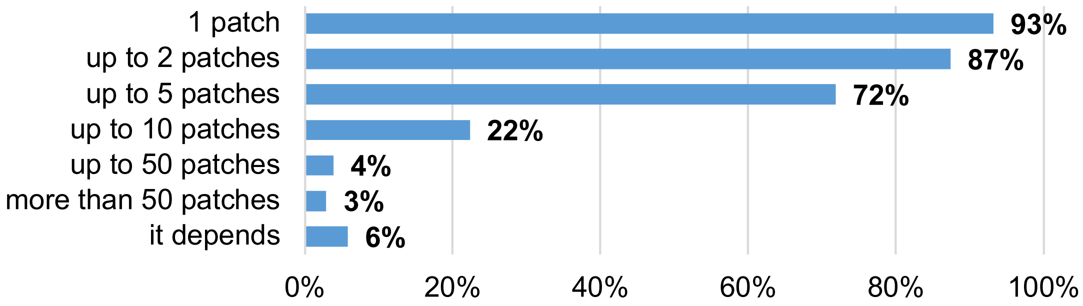


Figure 3.5: Cumulative illustration of the responses for Q1.2 *How many auto-generated patches would you be willing to review before losing trust/interest in the technique?*

but our results indicate that direct support of manual bug fixing requires a *quick* fix suggestion or hints. In fact, 82% of the participants state that they usually spend not more than 2 hours on average to fix a bug, and hence, the APR techniques need to be fast to provide a benefit for the developer. To increase the trust in the generated patches, 80% agree that *additional artifacts* (e.g., test cases), which are provided as input for APR, are useful (see Q1.6 in Figure 3.4). As a consistency check, we asked a similar question at a later point (see Q3.1 in Figure 3.4), and obtained that even 85% agree that additional artifacts can increase their trust. The most mentioned *other mechanisms* to increase trust are the *extensive validation* of the patches with a test suite and static analysis tools (17/103), the actual *manual investigation* of the patches (10/103), the *reputation* of the APR tool itself (9/103), the *explanation* of patches (8/103), and the provisioning of additionally *generated tests* (7/103).

RQ1 – Acceptability of APR: Additional user-provided artifacts like test cases are *helpful* to increase trust in automatically generated patches. However, our results indicate that *full* developer trust requires a manual patch review. At the same time, *test reports* of automated dynamic and static analysis, as well as *explanations* of the patch, can facilitate the reviewing effort.

The responses for the explicit question about developers' envisioned engagement with APR tools (Q2.8) can be categorized into four areas: the extent of *interaction*, the type of *input*, the expected *output*, and the expected *integration* into the development workflow.

Interaction Most participants (71/103) mention that they prefer a rather *low* amount of interaction, i.e., after providing the initial input to the APR technique, there will be no further interaction. Only a few responses (6/103) mention the one-time option to provide more test cases or some sort of specification to narrow down the search space when APR runs into a timeout, or the generated fixes are not correct. Only 3 participants envision a high level of interaction, e.g., repeated querying of relevant test cases.

Input Most participants appear ready to provide *failing test cases* (22/103) or *relevant test cases* (20/103). Others mentioned that APR should take a *bug report* as input (15/103), which can include the stack trace, details of the environment, and execution logs. Some also mentioned that they envision only the provision of the bare minimum, i.e., the program itself or the repository with the source code (11/103).

Output Besides the generated patches, the most mentioned helpful output from an APR tool is *explanations* of the fixed issue including its *root cause* (9/103). This answer is followed by the requirement to present not only one patch but a *list of potential patches* (8/103). Additionally, some participants mentioned that it would be helpful to produce a comprehensive *test report* (6/103).

Integration The most mentioned integration mechanism is to involve APR smoothly in the *DevOps pipeline* (17/103), e.g., whenever a failing test is detected by the CI pipeline, the APR would be triggered to generate appropriate fix suggestions. A developer would manually review the failed test(s) and the suggested patches. Along with the integration the participants mentioned that the primary goal of APR should be to *save time* for the developers (8/103).

RQ1 – Interaction with APR: Developers envision a *low* amount of interaction with APR, e.g., by only providing initial artifacts like test cases. APR should *quickly* (within 30 min - 60 min) generate a *small* number (between 5 and 10) of patches. Moreover, APR needs to be *integrated* into the existing DevOps pipelines to support the development workflow.

3.3.2.2 Availability/Impact of Artifacts

In this section, we look more closely in the categories C2 and C3 to investigate which additional artifacts can be provided by developers, and how these artifacts

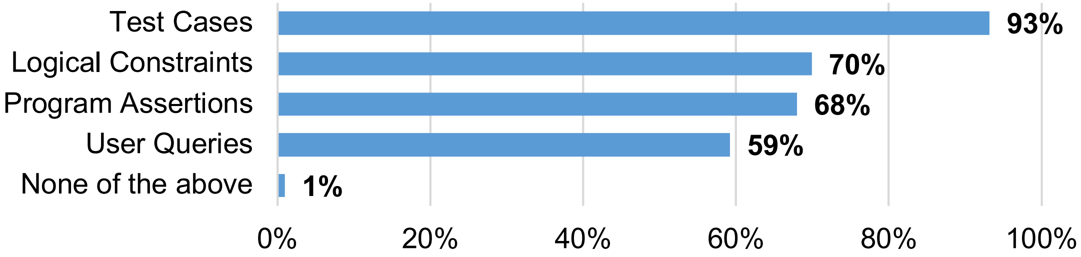


Figure 3.6: Responses for Q3.2 *Which of the following additional artifacts will increase your trust?*

influence the trust in APR. We first explore the availability of additional *test cases* (69% positive), *program assertions* (71% positive), and *logical constraints* (59% positive) (see the results for Q2.1, Q2.2, and Q2.3 in Figure 3.4). Furthermore, 58% of the participants are positive about answering queries to classify generated tests as failing or passing. This can be understood as participants want to have low interaction (i.e., asking questions to the tool), but if the tool offers to provide queries, they are ready to answer some of them (typically respondents preferring to answer no more than 10 queries). Based on the results for open-ended question Q2.7, majority of the participants (70/103) do not see any other additional artifacts (beyond tests/assertions/logical-constraints/user-queries) that they could provide to APR. The most mentioned responses by other participants are different forms of *requirements specification* (7/103), e.g., written in a domain-specific language, *execution logs* (6/103), documentation of *interfaces* with data types and expected value ranges (5/103), *error stack traces* (4/103), relevant *source code locations* (3/103), and reference solutions (3/103), e.g., existing solutions for similar problems.

RQ2 – Artifact Availability: Software developers can provide additional artifacts like test cases, program assertions, logical constraints, execution logs, and relevant source code locations.

On the *increase* of trust in patches by the incorporation of additional artifacts driving repair, 93% of the participants agree that additional *test cases* are helpful (Figure 3.6). This is also interesting from the perspective of recent automated repair tools [164] which perform automated test generation to achieve less overfitting patches. Logical constraints (70%) and program assertions (68%) perform worse in this respect. Although user queries allow more interaction with the APR technique, they would not necessarily increase the trust more than the other artifacts, as only 59% agreed on their benefit. Most of the participants (88/103) did not mention a trust gained by other artifacts. However, a notable artifact has been non-functional requirements (3/103) like performance or security aspects, which is related to a concern that auto-generated patches may harm existing performance characteristics or introduce security vulnerabilities.

RQ2 – Impact on Trust: Additional *test cases* would have a great impact on the trustworthiness of APR. There exists the possibility of automatically generating tests to increase trust in APR.

3.3.2.3 Patch Explanation/Evidence

In this section, we explore which patch evidence and APR side-products can support trust in APR (see categories C4 and C5). We first proposed two possible pieces of evidence that could be presented along with the patches: the *code coverage* achieved by the executed test cases that are used to construct the repair, and the *ratio of input space* that has been successfully tested by the automated patch validation. 76% of the participants agree that code coverage would increase trust, and 71% agree with the input ratio (see Q4.1 and Q4.2 in Figure 3.4). The majority of the participants (78/103) do not mention other types of evidence that would help to select a patch with confidence. Nevertheless, the most mentioned response is a *fix summary* (10/103), i.e., an explanation of what has been fixed including the root cause of the issue, how it has been fixed, and how it can prevent future issues. Other participants mention the *success rate* in case of patch transplants (5/103), and a *test report* summarizing the patch validation results (3/103). These responses match the observations for RQ1, where we asked how developers want to interact with trustworthy APR and what output they expect.

RQ3 – Patch Evidence: Software developers want to see *evidence* for the patch's correctness to efficiently select patch candidates. Developers want to see information such as code coverage as well as the ratio of the covered input space.

A straightforward way to provide explanations and evidence is to provide outputs that are already created by APR as side-products. We listed some of them

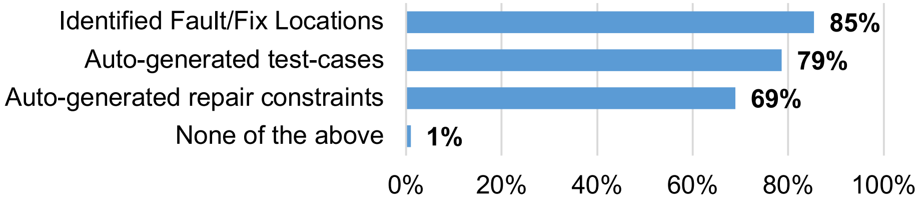


Figure 3.7: Responses for Q5.1 *Which of the following information (i.e., potential side-products of APR) would be helpful to validate the patch?*

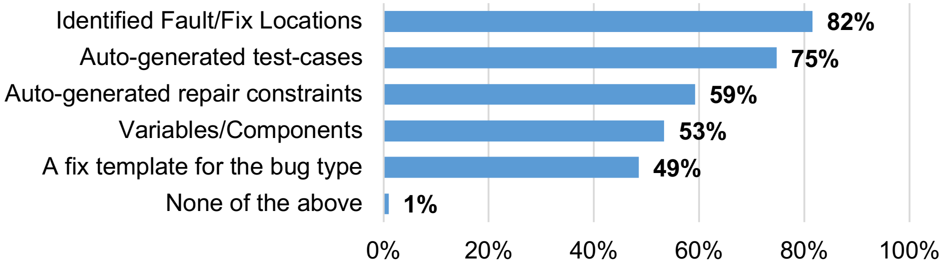


Figure 3.8: Responses for Q5.3 *Which of the following information (i.e., potential side-products of APR) would help you to fix the problem yourself (without using generated patches)?*

and asked the participants to select which of them would be helpful to validate the patches (see results in Figure 3.7). 85% agree that the identified *fault* and *fix locations* are helpful to validate the patch followed by the *generated test cases* with 79% agreement. In addition, a few participants emphasize the importance of a *test report* (4/103), an explanation of the *root cause* and the *fix attempt* (4/103).

Finally, we explore which side-products are most useful for developers, even when APR cannot identify the correct patch. Figure 3.8 shows that the identified fault and fix locations are of most interest (82%), followed by the generated test cases (75%). Very few participants add that an issue summary (2/103) and the potential results of a data flow analysis (2/103) could be helpful as well.

RQ3 – APR's Side-Products: Our results indicate that side-products of APR like the *fault* and *fix locations* and the *generated test cases* can assist manual patch validation, and hence, enhance trust in APR.

3.3.3 Threats to Validity

External Validity

Although we reached out to different organizations in different countries, we cannot guarantee that our survey results can be generalized to all software developers. To mitigate this threat, we made all research artifacts publicly available so that other researchers and practitioners can replicate our study. To reduce the risk of developers not participating or the volunteer bias, we designed the survey for a short completion time (15-20 min) and provided incentives like charity donations and (in the case of MTurk) monetary compensations.

Construct Validity

In our survey, to encourage candid responses from participants, we did not collect any personally identifying information. Additionally, we applied control questions to filter non-genuine answers. To mitigate the risk of wrong interpretation of the collected responses, we performed qualitative analysis coding, for which all codes have been checked and agreed by at least two authors. Although we found general agreement across participants for many questions, we consider our results only as a first step towards exploring trustworthy APR.

Internal Validity

Our participants could have misunderstood our survey questions, as we could not clarify any particulars due to the nature of online surveys. To mitigate this threat, we performed a small pilot survey with five developers, in which we asked for feedback about the questions, the survey structure, and the completion time. Additionally, there is a general threat that participants could submit multiple responses because our survey was completely anonymous.

3.4 Quantitative Evaluation of APR

We now investigate to which extent existing APR techniques support the expectations and requirements collected with our survey. Not all aspects of our developer survey can be easily evaluated. For example, the evaluation of the amount of interaction, the integration into existing workflows, the output format for the efficient patch selection, and the patch explanations, require additional case studies and further user experiments. In this evaluation, we focus on the quantitative evaluation of the relatively short patching time (30-60 min), the limited number of patches to manually investigate (5 to 10), handling of additional test cases and logical constraints, and the ability to generate a repair at a provided fix location. We explore whether state-of-the-art repair techniques can produce correct patches under configurations that match these expectations and requirements. Specifically, we aim to provide answers to the research questions RQ4 and RQ5.

3.4.1 Experimental Setup

3.4.1.1 APR Representatives

In our evaluation, we use the following representative state-of-the-art repair techniques: GenProg [68], Angelix [92], Prophet [79], and Fix2Fit [39]. GenProg [68] is a search-based program repair tool that evolves the buggy program by mutating program statements. It is a well-known representative of the generate-and-validate repair techniques. Angelix [92] is a semantic program repair technique that applies symbolic execution to extract constraints, which serve as a specification for subsequent program synthesis. Prophet [79] combines search-based program repair with machine learning. It learns a code correctness model from open-source software repositories to prioritize and rank the generated patches. Fix2Fit [39] combines search-based program repair with fuzzing. It uses grey-box fuzzing to generate additional test inputs to filter overfitting patches that crash the program. The test generation prioritizes tests that refine an equivalence class based patch space representation.

3.4.1.2 Subject Programs

We use the ManyBugs [46] benchmark, which consists of 185 defects in 9 open-source projects. For each subject, ManyBugs includes a test suite created by the original developers. Note that all of the studied repair techniques require and/or can incorporate a test suite in their repair process. For our evaluation, we filter the 185 defects that have been fixed by the developer at a single fix location. We remove defects from "Valgrind" and "FBC" subjects due to the inability to reproduce the defects. Finally, we obtain 60 defects in 6 different open-source projects (Table 4.1).

Table 3.3: Experiment subjects for quantitative evaluation of APR

Program	Description	LOC	Defects	Tests
LibTIFF	Image processing library	77k	7	78
lighttpd	Web server	62k	2	295
PHP	Interpreter	1046k	43	8471
GMP	Math Library	145k	1	146
Gzip	Data compression program	491k	3	12
Python	Interpreter	407k	4	355

3.4.1.3 Experimental Configurations and Setup

All tools are configured to run in full-exploration mode; which will continue to generate patches even after finding one plausible patch until the timeout or the completion of exploring the search space. To study the impact of fix locations and test case variations (see RQ5), we evaluate each tool using different configurations (see Table 3.4). Note that in each configuration we provide the relevant source file to all techniques, however, with "developer fix location" we provide the exact source line number as well.

Table 3.4: Experiment configurations for quantitative evaluation of APR

ID	Fix Location	Passing Tests	Timeout
EC1	tool fault localization	100%	1hr
EC2	developer fix location	100%	1hr
EC3	developer fix location	0%	1hr
EC4	developer fix location	50%	1hr

3.4.1.4 Evaluation Metrics

In order to assess the techniques, we consider eight metrics: **M1** the search space *size* of the repair tool, **M2** the number of *enumerated/explored* patches, **M3** the explored *ratio* with respect to the search space, **M4** the number of *non-compilable* patches, **M5** the number of *non-plausible* patches, i.e., patches that have been explored but ruled out because existing or generated test cases are violated, **M6** the number of *plausible* patches, **M7** the number of *correct* patches, and **M8** the highest *rank* of a correct patch. M1-M6 help to analyze the overall search space creation and navigation of each technique. The definition of the search space size (M1) for the defect, as well as the definition of an enumerated/ explored patch (M2), vary for each tool. We include all experiment protocols in our replication artifact, which describes how to collect these metrics for each tool. M7-M9 assess the repair outcome, i.e., the identification of the *correct* patch. We define a patch as *correct* whenever it is *semantically equivalent* to the developer patch from our benchmark. To check for the correct patch, we manually investigated only the top-10 ranked patches because our survey concluded that developers would not explore beyond that. Note that not all techniques provide a patch ranking (e.g., Angelix, GenProg, and Fix2Fit). In these cases, we use the order of generation as ranking.

3.4.1.5 Hardware

All our experiments were conducted using Docker containers on top of AWS (Amazon Web Services) EC2 instances. We used the c5a.8xlarge instance type, which provides 32 vCPU processing power and 64GiB memory capacity.

3.4.1.6 Replication

Our replication package contains all experiment logs and subjects, as well as protocols that define the methodology used to analyze the output of each repair tool.

3.4.2 Evaluation Results

Table 3.5 summarizes our evaluation results. For each APR technique we show its performance under the given experimental configuration (see Table 3.4). Each cell shows $|P_{Plaus}|/|P_{Corr}|$, where $|P_{Plaus}|$ is the number of defects for which the tool was able to generate at least one plausible patch (i.e., M6), and similarly $|P_{Corr}|$ is the number of defects for which the tool was able to generate a correct patch among the top-10 plausible patches. For example, the LibTIFF project has 7 defects, for which Angelix was able to generate 3 plausible and 1 correct patch for the setup EC1 (i.e., 1-hour timeout, tool fault localization, and all available test cases). Due to limitations in its symbolic execution engine Klee [16], Angelix does not support lighttpd and python, and the corresponding cells are marked with “-”. Additionally, Table 3.6 presents the average patch exploration/enumeration ratio $|P_{Expl}|$ of the techniques with respect to the patch space size, computed as a percentage of M2/M1 for each defect considered in each subject.

3.4.2.1 APR within realistic boundaries

The numbers in Table 3.5 show that the overall repair success is comparably low. For example, Fix2Fit can generate plausible patches for 14 defects with EC1. Compared to previous studies, the number of plausible patches is significantly lower in our experiments, mainly due to the 1-hour timeout. Prior research on program repair have experimented with 10-hours [89], 12-hours [92, 79] and 24-hours [39] timeouts, and determine if a correct patch can be identified among *all* generated plausible patches. The focus of these prior experiments was to evaluate the capability to generate a patch, whereas, in our work, we focus on the performance within a tolerable time limit set by developers.

Table 3.5: Quantitative evaluation of APR results for the various configurations

Subject	**Def.**	**Angelix**				**Prophet**				**GenProg**				**Fix2Fit**			
		EC1	EC2	EC3	EC4	EC1	EC2	EC3	EC4	EC1	EC2	EC3	EC4	EC1	EC2	EC3	EC4
LibTIFF	7	3/1	3/1	3/1	3/1	1/0	1/0	1/0	1/0	5/0	5/0	5/0	5/0	5/1	4/1	4/1	4/1
lighttpd	2	-	-	-	-	1/0	0/0	0/0	0/0	1/0	1/0	1/0	1/0	1/0	1/0	1/0	1/0
PHP	43	0/0	0/0	0/0	0/0	0/0	0/0	2/1	3/1	0/0	0/0	10/1	0/0	8/1	4/2	7/2	5/1
GMP	1	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0
Gzip	3	0/0	1/0	1/0	1/0	0/0	1/1	1/1	1/1	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0
Python	4	-	-	-	-	0/0	1/1	1/1	1/1	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0
Overall	60	3/1	4/1	4/1	4/1	2/0	3/2	5/3	6/3	6/0	6/0	16/1	6/0	14/2	9/3	12/3	10/2

Each cell shows the number of subjects, for which the technique was able to identify at least one *Plausible/Correct* patch with regard to the specific configuration.

RQ4 – Repair Success: Current state-of-the-art repair techniques perform poorly with a 1-hour timeout and the top-10 ranking restriction. Most techniques cannot identify any plausible patch for most defects in the ManyBugs benchmark.

In general, the repair success of an APR technique is determined by (1) its search space, (2) the exploration of this search space, and (3) the ranking of the identified patches. In a nutshell, this means, if the correct patch is not in the search space, the technique cannot identify it. If the correct patch is in the search space, but APR does not identify it within a given timeout or other resource limitations, it cannot report it as a plausible patch. If it identifies the patch within the available resources but cannot pinpoint it in the (potentially huge) space of generated patches, the user/developer will not recognize it. By means of these impediments for repair success in real-world scenarios, we examine the considered repair techniques. Our goal is to identify the concepts in APR that are necessary to achieve the developers' expectations, and hence, to improve the state-of-the-art approaches.

Search Space Table 3.6 shows that Angelix explores almost its complete search space within the 1-hour timeout, while Table 3.5 shows that it can identify plausible patches for only one defect (with EC1). As described in [89], the program transformations (to build/explore the search space) by Angelix only include the modification of existing side-effect-free integer expressions/conditions and the addition of if-guards. Therefore, we conclude that Angelix's search space is too limited to contain the correct patches.

Table 3.6: Average exploration ratio $|\mathbf{P_{Expl}}|$ for EC1 and EC2.

Subject	**Angelix**		**Prophet**		**GenProg**		**Fix2Fit**	
	EC1	EC2	EC1	EC2	EC1	EC2	EC1	EC2
LibTIFF	86	100	24	93	1	27	100	100
lighttpd	-	-	20	100	<1	51	100	100
PHP	96	100	22	96	<1	91	63	80
GMP	100	100	41	100	5	100	-	-
Gzip	100	100	6	100	18	100	100	100
Python	-	-	14	100	1	100	-	-
Overall	95	100	21	98	4	78	91	95

The other techniques, on the other hand, consider larger search spaces. Prophet also considers the insertion of statements and the replacement of function calls. GenProg can insert/remove any available program statement. Fix2Fit uses the search space by f1x [89], which combines the search spaces of Angelix and Prophet to generate a larger search space.

RQ4 – Search Space: Successful repair techniques need to consider a wide range of program transformations and should be able to take *user input* into account to enrich the search space.

Search Space Exploration Prophet and GenProg show a low exploration ratio with 21% and 4% respectively (see EC1 in Table 3.6) that leads to a low number of plausible patches. Instead, Fix2Fit fully explores the patch search space for most of the considered defects (except for PHP), which leads to a high possibility of finding a plausible patch.

In contrast to Prophet and GenProg, Fix2Fit performs *grouping* by their behavior on test inputs and uses this equivalence relation to guide the generation of additional inputs. Prophet and GenProg, however, need to explore and evaluate all concrete patches, which causes a significant slowdown. Reduction of the patch validation time is possible if we can validate patches without the need to re-compile the program for each concrete patch [31, 23, 157].

RQ4 – Patch Space Exploration: A large/rich search space requires an *efficient* exploration strategy, which can be achieved by, e.g., using search space *abstractions.*

Patch Ranking Although Fix2Fit builds a rich search space and can efficiently explore it, it still cannot produce many correct patches. One reason is that Fix2Fit can identify a correct patch but fails to pinpoint it in the top-10 patches because it only applies a rudimentary patch ranking, which uses the edit-distance between the original and patched program. For instance, Fix2Fit generates the correct patch for the defect *865f7b2* in the LibTiff subject but ranks it below position 10, and hence, it is not considered in our evaluation. Furthermore, Fix2Fit's patch refinement and ranking is based on crash-avoidance, which is not suitable for a test-suite repair benchmark such as ManyBugs that does not include many crashing defects.

We also investigated how many of the correct patches are within the top-5 because 72% of our survey participants strongly favored reviewing only up to 5 patches (see Figure 3.5). We observed that most identified correct patches within the top-10 are ranked very high so that there is not much difference if a top-5 threshold is applied. Recent work [159, 157] propose to use test *behavior* similarity between original/patched programs to rank plausible patches.

RQ4 – Patch Ranking: After exploring the correct patch, an effective patch ranking is the last impediment for the developer.

3.4.2.2 Impact of additional inputs

Providing Fix Location as User input In Table 3.5, the column EC1 shows the results with the tool's fault localization technique, and column EC2 shows the results by repairing only at the developer-provided (correct) fix location. Intuitively, one expects that equipped with the developer fix location, the results of each repair technique should improve. However, the results by Angelix and GenProg do not change (except for one more plausible patch with Angelix). From the previous discussion about the search space, we conclude that the program transformations by Angelix are the main limiting factor so that even the provision of the correct fix location has no impact. For GenProg we know from the EC3 configuration that there is at least one correct patch in the search space (see Table 3.5). Therefore, we conclude that GenProg suffers from its inefficient space exploration so that even the space reduction by setting the fix location has no impact. Prophet instead can generate two additional correct patches in EC2, and hence, benefits from the precise fix location. The exploration ratio in Table 3.6 shows that Prophet almost fully explores its search space in EC2, indicating the significantly smaller search space. Fix2Fit can generate one more correct patch as compared to EC1. Similar to Prophet, Fix2Fit benefits from the precise fix location and can explore more of its search space.

RQ5 – Fix Location: Our results show that the provision of the precise and correct fix location does not necessarily improve the outcome of the state-of-the-art APR techniques due to their limitations in search space construction and exploration.

Varying Passing Test Cases To examine the impact of the passing test cases, we consider the differences between the columns EC2, EC3, and EC4 in Table 3.5. In general, more passing test cases can lead to high-quality patches because they represent information about the correct behavior. In line with this, we observe that more passing test cases lead to less plausible patches because the patch validation can remove more overfitting patches. For Angelix however, we observe that there is no difference due to its limited search space. Overall, we observe three different effects: (a) For techniques with a limited search space (e.g., Angelix), passing test cases have very low or no effect. (b) For techniques that suffer from inefficient space exploration strategies (e.g., GenProg and Prophet), having fewer passing test cases can speed up the repair process and lead to more plausible (possibly overfitting) patches. (c) Otherwise (e.g., Fix2Fit), variations in the passing test cases can still influence the ranking. Whether more tests are better depends on the APR strategy and its characteristics, as discussed in Section 3.4.2.1. Therefore, we suggest that APR techniques incorporate an intelligent test selection or filtering mechanism, which is not yet studied extensively in the context of APR. Recently, [81] suggested applying traditional regression test selection and prioritization to achieve better repair efficiency. Further developing and using such a mechanism represents a promising research direction. Note that in the discussed experiments, the fix location was defined beforehand. However, if APR techniques use a test-based fault localization technique (like in EC1), the test cases have an additional effect on the search space and repair success.

RQ5 – Test Cases: Variation of passing test cases causes different effects depending on the characteristics of the APR techniques. Overall, one needs an intelligent test selection method.

3.4.3 Threats to Validity

3.4.3.1 External Validity

In our empirical analysis, we do not cover all available APR tools, but instead, we cover the main APR concepts: search-based, semantics-based, and machine-learning-based techniques. With ManyBugs [46] we have chosen a benchmark that is a well-known collection of defects in open-source projects. Additionally, it includes many test cases, which are necessary to evaluate the aspects of test case provision.

3.4.3.2 Construct Validity

The metrics in our quantitative evaluation measure the patch generation progress, measuring repair efficiency/effectiveness via variations in configurations (EC1-EC4).

3.4.3.3 Internal Validity

To mitigate the threat of errors in our setup of the empirical experiments, we performed preliminary runs with a subset of the benchmark and manually investigated the results.

3.5 Summary

In this chapter, we have investigated the issues involved in enhancing developer trust in automatically generated patches. Through a detailed study with more than 100 practitioners, we explore the expectations and tolerance levels of developers with respect to automated program repair tools. We then conduct a quantitative evaluation to show that existing repair tools do not meet the developer's expectations in terms of producing high-quality patches in a short time period. Our qualitative and quantitative studies indicate directions that need to be explored to gain developer trust in patches: low interaction with repair tools, exchange of artifacts such as generated tests as inputs as well as output of repair tools, and paying attention to abstract search space representations over and above search algorithmic frameworks.

Chapter 4

Accelerating Vulnerability Repair via Compilation-Free Validation

Automated Program Repair (APR) aims to automatically repair software bugs without developer (human) intervention. Over the years, many different APR tools and techniques have been proposed. These include program repair based on symbolic execution, machine learning, as well as search based repair. For search-based or *Generate and Validate (G & V)* based repair, the APR tool searches for and generates a set of candidate patches at a set of fix locations, subsequent to which these patches are validated and ranked using a oracle, such as a user-provided test suite. The search-based repair methodology is popular, and has been implemented by several prominent APR tools for `C/C++`. G&V-based repair known to suffer from some practical limitations at the *validation* step in which requires the candidate patch be applied to the program and *recompiled*, before tested against the test suite. However, recompilation can be time consuming, especially if the G&V repair tool needs to validate thousands of potential patch candidates. In this chapter, we propose to accelerate G&V-based repair by essentially removing the compilation step, thereby achieving the goal of *Compilation Free program Repair* (CFR). Using a combination of *binary rewriting* and patch *interpretation*, we directly validate candidate patches "on-the-fly" without the need for program recompilation. We show that CFR can significantly improve the performance of G&V-based repair, with lower validation latency and much higher validation throughput. This would allow for navigation of significantly larger search spaces for program repair within a time limit which is tolerable to developers. Furthermore, a generic CFR system can concurrently support multiple program repair tools, and can be implemented using a standard (off-the-self) tool-chain such as `binutils`/GDB.

4.1 Overview

In this chapter, we explore a perennial problem in automated program repair, which hinders the validation step in the repair process, limiting the efficiency of the overall repair process. The general APR methodology works by automating various steps in the typical (manual) debugging workflow, including: *fix-localization* (i.e., *where* to apply the fix?), *patch-generation* (i.e., *how* to fix the bug?), and *patch-validation/ranking* (i.e., *validate* that the patch correctness). Many APR tools are based on a *Generate-and-Validate* (G&V) methodology. Here, the basic approach is to treat APR as a search problem over the space of potential program edits, with the idea that the "correct" patch/fix must be an element of this space. One known issue in this approach is that the cost of patch validation can be a significant bottleneck [23]. For compiled programming languages, such as `C`/`C++`, the patch must be first *applied* to the program and *recompiled* before it can be validated against a test suite. However, recompilation (including relinking) can be a relatively expensive operation, possibly in the order of seconds or minutes, depending on the size of the program. This problem can severely limit both the *latency* (i.e., time to identify a plausible patch) and *throughput* (i.e., number of patches validated per time budget) of G&V-based repair. As such, many G&V-based APR tools are *offline*, meaning that they are intended to be run over several hours, such as overnight, in order to find useful patch candidates. However, the recent work of [101] shows that most real-world developers expect answers from APR tools in much shorter time frames, with 72% of survey respondents preferring not even to wait longer than 30 minutes. We therefore argue that the *latency* of repair tools is critically important, especially for real-world adoption, and is something that is largely neglected by most existing APR research. A low latency design means that APR tools should validate patches promptly—as soon as patch candidates are generated—providing the developer with continuous feedback as new candidates are validated.

One way to improve the latency/throughput is to remove the bottlenecks, such as recompilation, from G&V-based repair. To do so, we propose to replace recompilation with a combination of *interpretation* and *binary rewriting/probing*, in order to remove the *compiler-in-the-loop* from G&V-based repair—i.e., *Compilation Free Repair* (CFR). Compared to recompilation, an interpreter can be low latency with a minimal startup time, allowing for patches to be validated immediately upon generation. Since whole-program interpretation is slow, our proposal uses *binary rewriting/probing* to limit the interpreted expressions/statements to those actually changed by the patch, leaving the rest of the program to use native (compiled) execution. We show that compilation free repair can significantly improve the latency/throughput of G&V-based repair tools, by an order of magnitude.

In addition to latency/throughput improvements, we show that a CFR-based system has other benefits. We can design CFR to be *generic*, meaning that it can operate independent of the patch generatation frontend(s). In other words, a single CFR system can drastically improve the efficiency of a variety of program repair tools, as also shown in our experiments. To do so, we design a CFR-based backend that accepts patches represented in the standard, human-readable, Unix `diff` format; the patches are generated by one or more G&V frontends. The CFR backend will parse each generated candidate patch, and validate/rank it against the given test oracle. The generic design makes it easy to port existing G&V frontends, integrate new patch generation frontends, and to support mixing-and-matching of different frontends which can run concurrently. In contrast, existing G&V repair tools tend to implement their own specialized validation solutions. We argue that this is duplicated effort, and we can consolidate patch validation/ranking into a single shared framework.

The final technical challenge for CFR is the implementation. We note that implementing a fully featured interpreter for mature languages, such as `C/C++`, can be a non-trivial effort. To this end, we show that CFR can be implemented on top of standard tools, namely `gdb`, which have a stable/mature interpreter already built-in. We present a prototype CFR implementation which translates patch candidates into an equivalent `gdb`-script. The test case(s) can be run under `gdb`, and the script will intercept and execute the modified patch statements. We show that our approach is low-latency and high-throughput, with minimal overheads.

4.2 Motivation

Automated Program Repair (APR) attempts to patch buggy programs without manual intervention from programmer. Over the past decade of research, many different APR tools and technologies have been developed, including those based on *machine-learning*, *constraint-solving*, *templates*, and *search*-based methods. Many APR tools are based on a *Generate-and-Validate* (G&V) methodology, which attempts to *generate* a set of candidate patches and, then *validate* or *rank* the patches according to some (possibly incomplete) oracle, such as a test-suite. The *validation* step requires a patched version of the program to be built so that it can be evaluated (against the test-suite). For compiled languages such as `C/C++`, this is achieved by *recompiling* program with the patch applied.

Performance One known problem with the G&V-based repair is the cost of the *recompilation.* Depending on the size of the target program, recompilation (including re-linking), may take in the order of a few milliseconds up to several minutes for very large projects. The recompilation costs are the source of significant overhead for G&V-based repair. Even if the cost of each recompilation is minimized, a typical G&V program repair tool will need to validate thousands of patches, meaning that the costs can quickly add up. For example Prophet spent 56.9 minutes for recompilation when attempting to repair CVE-2017-15020 vulnerability in binutils, within a time-bound of 1 hour, which amounts to 94.8% of total time. Overall, a single recompilation step takes an average of 2 seconds per compilation step for Prophet to repair CVE-2017-15020, but the total cost gets accumulated over 1637 candidate patch enumerations. Thus, recompilation is the main performance bottleneck in G&V based automated program repair. To illustrate further, figure 4.1 depicts the average time spent in recompilation by the prominent repair tools Prophet and Darjeeling for subjects in the VulnLoc benchmark [122] of vulnerabilities. Overall repair time is constrained to 1 hour, which clearly shows a significant time from the allocated budget (50% or more of the 1 hour time budget) is spent on recompilation alone. Note that the time spent on recompilation depends on the search-space of the repair tool, hence a performance comparison across the tools based on the recompilation time alone is not meaningful.

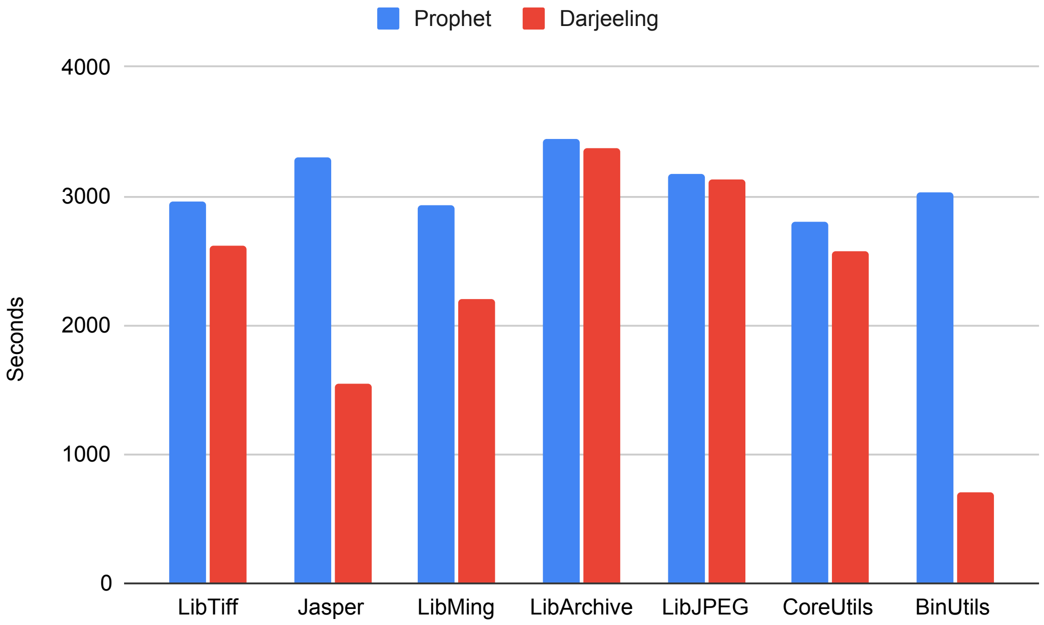
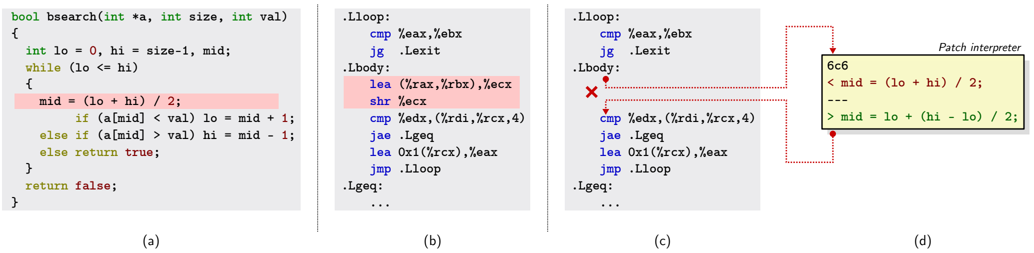


Figure 4.1: Average time spent for recompilation by Prophet and Darjeeling for subjects in VulnLoc benchmark

The recent work of [101] surveyed over 100 developers regarding their expectations in relation to automated program repair. The results showed that most developers expect APR tools to provide answers promptly, with 72% of survey respondents preferring to wait no longer than 30 minutes, with the concensus being that faster is always better. In contrast, most existing APR literature evaluates tools using a more generous fixed time budget, with 10/12/24 hours being typical. If we need to run a repair tool for 10-24 hours, its practical usage becomes limited — such as running the APR tool overnight in the hope of finding a fix the next day—i.e., essentially a form of *offline* repair. The results of the survey [101] suggest that it is essential for APR tools provide answers promptly in order to enjoy more mainstream real-world adoption. We therefore believe that it is necessary to evaluate the *latency* of APR solutions (i.e., the how fast the APR tool can generate a patch?) in addition to the performance over a fixed time budget (*throughput*).

Optimizations The high cost of recompilation in G&V-based repair has been recognised previously, and several optimizations have been proposed. For example, Prophet [79] attempts to optimize the recompilation process by deferring the instantiating of some conditions to validation time, thereby minimizing recompilation. However, this does not remove the compiler-in-the-loop entirely, which still incurs significant time overhead. Another optimization is *super mutant* recompilation, which attempts to batch several patches into a single (i.e., "super") patch, which can be controlled during validation/test time, e.g., by using a controllable switch. This is essentially a form *batched* recompilation, meaning that instead of recompiling N times, we can recompile once using a super mutant. While this does reduce recompilation, it is not entirely eliminated. Furthermore, batched recompilation means that the patch validation until the batch is generated, which can degrade latency. Finally, Darjeeling [145] also attempts to minimize recompilation costs by using *parallel recompilation* (e.g., running multiple virtual containers). However, this approach can only improve performance under the assumption that some non-utilized CPU resource (i.e., additional cores) is available. Assuming full utilization, parallel (re)compilation offers no additional benefit.

Our focus is therefore how to efficiently validate/rank general patches without resorting to specialized solutions. *Specialized solutions* to mitigate compilation costs, are somewhat ad-hoc, where each tool implements its own (optimized) patch validation solution that does not necessarily generalize, resulting in duplicated effort. Instead, we argue that the concept of test-based patch validation is *generic*, meaning that, with the appropriate engineering, a single patch *validation* backend can support multiple patch *generation* frontends, where each frontend can even support differing underlying methodologies. Rather than specialized optimizations, we instead propose compilation-free repair (CFR) as a *generic* optimization that can benefit any frontend, regardless of the how the frontend works or how it is implemented. Furthermore, patch candidates can be generated by many different methods, including *mutation*, *transplantation*, *machine learning* techniques, or through *program synthesis*. It is possible that one method will perform better on some classes of repair problems, but not others. Rather than expecting the user to try multiple tools (i.e., *trial-and-error*), a unified validation frameworks makes it possible to deploy multiple patch generation frontends at the same time, of which the best-ranked patch(es) can be presented and selected by the end-user. We can thus consolidate patch validation into a single framework which is shared by multiple G&V tools, both present and future ones.



```
bool bsearch(int *a, int size, int val)
{
  int lo = 0, hi = size-1, mid;
  while (lo <= hi)
  {
    mid = (lo + hi) / 2;
         if (a[mid] < val) lo = mid + 1;
    else if (a[mid] > val) hi = mid - 1;
    else return true;
  }
  return false;
}
```

(a)

```
.Lloop:
    cmp %eax,%ebx
    jg  .Lexit
.Lbody:
    lea (%rax,%rbx),%ecx
    shr %ecx
    cmp %edx,(%rdi,%rcx,4)
    jae .Lgeq
    lea 0x1(%rcx),%eax
    jmp .Lloop
.Lgeq:
    ...
```

(b)

```
.Lloop:
    cmp %eax,%ebx
    jg  .Lexit
.Lbody:

    cmp %edx,(%rdi,%rcx,4)
    jae .Lgeq
    lea 0x1(%rcx),%eax
    jmp .Lloop
.Lgeq:
    ...
```

(c)

```
6c6
< mid = (lo + hi) / 2;
---
> mid = lo + (hi - lo) / 2;
```

(d)

Figure 4.2: An illustration of *patch interpretation*. Here, (a) is an implementation of binary-search which contains an integer overflow bug (highlighted), (b) is the program compiled into assembly, with the instructions corresponding to the buggy line also highlighted. Both (c) and (d) illustrate the implementation of patch interpretation. Here, the instructions corresponding to the buggy line (c) are replaced with a call to the patch interpreter. The interpreter executes the replacement line (d) before returning control-flow back to the main program.

4.3 Compilation-Free Repair

The aim of *Compilation-Free Repair* (CFR) is to remove the *recompilation* bottleneck of (G&V)-based program repair. To do so, our overall approach is to replace compilation with low-latency alternatives, specifically an *interpreter* that can execute patch statements/expressions. Furthermore, we show that CFR can be made *generic*, meaning that a single patch *validation* backend can support multiple patch *generation* frontends, which can run concurrently and be mixed-and-matched. In this section, we outline the design for our CFR framework. In the next section, we shall present a practical implementation based on standard tools (`binutils`).

4.3.1 Removing the Compiler-in-the-Loop

Existing G&V APR tools for `C/C++` programs rely on recompilation to create patched binaries for testing. Modern compilers, such as `gcc` and `clang`, use a complex multi-stage pipeline, including *parsing*, *analysis*, *optimization*, *emission*, *linking*, etc., in order to optimize the runtime performance of the resulting executable. Although compiler analysis and optimization can be expensive, this is usually a once-off investment, since the goal is to emit a compiled program that will be executed some arbitrary number of times. Thus, the initial time investment for compilation is amortized over the lifetime of compiled program, and is justifiable. However, in the case of G&V-based repair, the compiled program is only executed on a few test cases to validate a patch, meaning that compilation costs can be excessive. Even when disabling optimization (with `-O0`), the (re)linking costs alone can be prohibitive for this use case.

An alternative to compilation is *interpretation*, which aims to execute the program without compilation into native machine code. A basic interpreter can use a short pipeline consisting of just *parsing* and *evaluation*, meaning that statements/-expressions can be executed "immediately" with minimal delay, i.e., without waiting for costly analysis, optimization and linking. This can be useful for applications where latency is important. For example, `gdb`'s (`print` *expr*) command needs evaluate the `C/C++` expression (*expr*) in an interactive environment. Thus, the expression will be evaluated immediately using `gdb`'s built-in interpreter, rather relying on a compiler. That said, the main disadvantage is that an interpreted program can run significantly slower than the compiled equivalent, sometimes by orders of magnitude.

For the G&V-based repair use case, it is also important to minimize the latency so that candidate patches can be evaluated immediately, without waiting for recompilation. Furthermore, it is also necessary to not to slow down testing too much, meaning that whole-program interpretation is not a practical alternative. Instead, we proposal is to use a hybrid approach, where a low-latency interpreter is used to execute the expressions/statements changed by the patch—i.e., *patch interpretation*—and native (compiled) code is used for everything else.

Patch interpretation An example illustration of patch interpretation is shown in Figure 4.2. Here, we use a buggy implementation *binary search* (Figure 4.2 (a)), which is used to test the membership of a (`val`) in a sorted array (a). The binary search algorithm works by repeatedly narrowing a range (`lo`..`hi`) of array indices, until either `val` is found (success) or the range becomes empty (failure). Each iteration calculates the midpoint of the range `mid`=(`lo`+`hi`)÷2. However, the naïve method for calculating the midpoint is well-known to be vulnerable to an *integer overflow* error. Specifically, the sub-expression (`lo`+`hi`) may exceed the maximum integer value (`INT_MAX`). The bug can be fixed by using an overflow-safe method for calculating the midpoint, as expressed by the following patch:

```
6c6
< mid = (lo + hi) / 2;
---
> mid = lo + (hi - lo) / 2;
```

Here, the patch file uses the standard Unix `diff` format, where the (`6c6`) indicates the location (line 6) that is replaced, the (`<`) identifies the original line to be deleted, and the (`>`) is the replacement line to be added. The replacement calculates the midpoint using the difference ($\texttt{hi}-\texttt{lo}$), which cannot exceed the `hi` value for $\texttt{lo}\leq\texttt{hi}$, thereby avoiding any potential integer overflow.

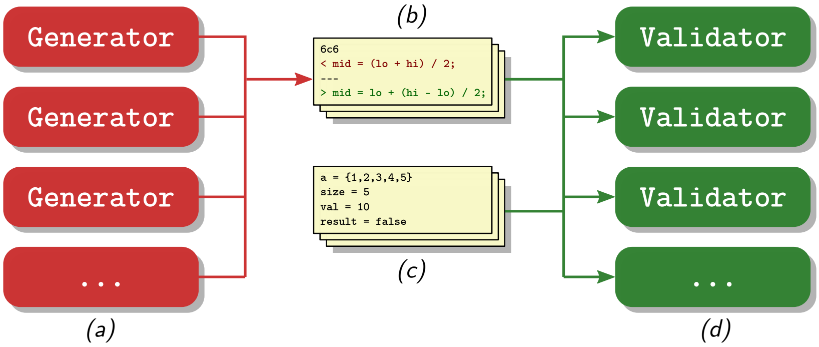
Under *patch interpretation*, we assume that the program has already been compiled into a binary executable B, as (partially) shown in Figure 4.2 (b). Here, the instructions corresponding to the buggy line from Figure 4.2 (a) have also been highlighted. Given a patch P, *patch interpretation* works by

1. *Diverting* control-from to/from the patched program locations, as identified by the deleted line(s) (`<`) from the patch file P; and

2. *Interpreting* the patched statements/expressions, as identified by the replacement line(s) (`>`) from the patch file P.

An example of patch interpretation is illustrated in Figures 4.2 (c) and (d). Here, the program will execute natively until the patch location is reached, as identified by the deleted line (`<`) from the patch file. Next, control-flow is diverted to a low-latency *interpreter*, which interprets the replacement line (`>`). Finally, control-flow is returned back to the original binary and native execution resumes.

Implementing CFR via patch interpretation faces several implementation challenges, such as: how to find the instructions corresponding to source lines of code? how to read and write to source-level variables (`lo`, `hi`, `mid`, etc.)? how to divert control-flow to-and-from the patch interpreter? For these challenges, we will present (in Section 6.5) a practical solution based on the compiler-generated debug information in DWARF format (using the compiler `-g` option).



```
6c6
< mid = (lo + hi) / 2;
---
> mid = lo + (hi - lo) / 2;
```

```
a = {1,2,3,4,5}
size = 5
val = 10
result = false
```

Figure 4.3: Basic workflow illustration. Here, one (or more) patch generation *frontends* (a) generate candidate patches which are collected into a *patch pool* (b). Next, one (or more) patch validation *backends* (d) validate pooled candidate patches against the test suite (c) using CFR. The frontends and backends are independent processes, and can run concurrently.

4.3.2 A Compilation-Free Framework

Patch interpretation forms the basis for CFR, allowing for patches to be validated, without delay, as soon as they are generated. Furthermore, patch interpretation accepts a (generic) Unix `diff` format as input, meaning that patch candidates from any source can be validated using the same backend, allowing for a *compilation-free repair framework*.

Workflow The basic workflow of the CFR framework is shown in Figure 4.3. The framework consists of one (or more) *frontends* (Figure 4.3 (a)) that generate patch candidates, and are used to populate a *patch pool* (Figure 4.3 (b)). The generated patch candidates are then validated by one (or more) *backends* (Figure 4.3 (d)) using the provided test suite (Figure 4.3 (c)). Under this workflow, each frontends and backends are independent processes, and communicate only via the *patch pool*. Each backend process validates candidate patches using the *patch interpretation* method as illustrated in Figure 4.2.

The Figure 4.3 workflow is designed to be *generic*. Since the patch interpretation backend accepts standard Unix `diff` format as input, the patch generation frontends need only to support this format in order to be compatible. As will be shown, we can readily adapt several existing G&V-based repair tools (namely Prophet, and Darjeeling) to emit patches in Unix `diff` format. This design also has the advantage of *consolidation*. Previously, each individual G&V tool will implement their own, specialized, validation solution. However, this is essentially duplicated effort, and we can show that a consolidated backend support multiple G&V-based repair tools, and with better performance.

Finally, the Figure 4.3 workflow is designed to be *concurrent*. Each frontend can be viewed as a patch *source* (producer), and each validation backed can be viewed as a patch *sink* (consumer), and the frontends/backends only communicate/synchronize via the patch pool. This allows for each frontend and backend instance to run as separate (concurrent) processes, which can further improve the overall performance. Furthermore, this design also allows for more than one backend instance to run concurrently, which can boost the overall validation bandwidth. Finally, it is also possible to run multiple (different) patch generation frontends at the same time, essentially making it possible to "mix-and-match" compatible G&V-based APR tools. Such a configuration can be advantageous for the case when different frontends have different strengths and weaknesses, depending on the class of bug.

4.4 Implementation

Implementing CFR faces several technical challenges, such as how to find the patch location/variables? how to divert control-flow to and from the interpreter? Etc. Furthermore, implementing a fully featured interpreter for a modern language, such as `C/C++`, is a non-trivial effort. To address these challenges, in this section we present a baseline implementation of CFR that is built on top a standard tool, specifically the `gdb` debugger.

The first step in our implementation is to is to convert a patch file generated by the repair back-end into a executable script. Since the input for the patch validator is a file in Unix diff format, identifying control-flow locations based only on the textual change, is a challenge, hence we make use of the original source-file in combination with the patch-file to identify control-flow information. Source-file provides the necessary control-flow information for the interpreter to inject the modified behavior during the execution based on the changes in the patch file. For example consider a patch changing the conditional expression for a if statement, for which the Unix diff file will only provide the changed lines, however for the interpreter its also important to identify which locations to change control flow if the condition is not satisfied, which will be present in the original source file.

Once we embed control-flow information as meta-data into the source file, the next step is to convert a patch file generated by the repair back-end into a executable script. The basic idea is to *translate* each given candidate patch, represented in Unix `diff` format, into an equivalent `gdb`-*script* that implements the patch as a set of `gdb` commands.

For the example patch from Figure 4.2, an equivalent `gdb`-script is:

```
1   set pagination off
2   set disable−randomization off
3   set breakpoint pending on
4   break bsearch.c:6
5   commands
6   silent
7   set var mid = lo + (hi − lo) / 2
8   jump bsearch.c:7
9   end
```

In general we add initialization configurations for `gdb` applicable for all candidate patches, for example setting breakpoint pending to the value *on*, instructs `gdb` to append breakpoints that are not exclusive for the executing binary but may also be of a linked library. Next, the script instructs `gdb` to set a breakpoint at the patch location (`bsearch.c:6`). For each modified code segment in the source file a breakpoint will be added to inject the modified behavior at the specified location. For each breakpoint `gdb` takes a list of instructions that can be executed as a `gdb` command. A snippet of the schema is shown below:

lvalue `=` *expr*	`set var` *lvalue* `=` *expr*
`f`(...)	`call f`(...)
`return` *value*	`return` *value* + `continue`
`if` (*expr*) `{`...`}`	`if` (*expr*) ... `end`
`if` (*expr*) `{`...`} else {`...`}`	`if` (*expr*) ... `else` ... `end`

In order to handle loops, we insert two breakpoints at the beginning and end of the loop, to effectively control the iteration(s). In our example gdb script shown above, the commands specify that the `mid` variable should be updated with the patched expression (line 7), and that control-flow should resume after the patch location (`bsearch.c:7`) at line 8. For patches that modify control-flow such as if/while/for statements the control-flow will be updated using the embedded metadata provided in the annotated source-file. In our example, the `set` command will execute the patched assignment before control-flow is returned to the point immediately after the patch location. This process may repeat if control-flow reaches the patch location multiple times (e.g., in a loop). Since the `gdb set` command (and similar commands) use a low-latency interpreter internally, the `gdb`-script essentially implements patch interpretation without recompilation.

Final step is to execute the test case(s) and observe the output to determine if the candidate patch is passing or failing. We achieve this by simply replacing the binary file with a bash script that acts as the gdb-driver for the test execution which will take as argument the gdb script and the input for the test execution. The test is instantiated dynamically by appending the gdb command `run` into the `gdb` script, and invoking the `gdb` interpreter.

4.5 Evaluation

We evaluate our approach for compilation-free repair, by extending three prominent repair tools for C/C++ programs namely Prophet and Darjeeling. We modify these tools to be able to dump the search space once constructed, for CFR validation by disabling internal validation mechanisms. For this evaluation, we formulate the following research questions.

RQ1 How does CFR perform compared to existing recompilation optimizations?

RQ2 Can CFR benefit with parallelize validation to improve the throughput of existing G&V repair techniques?

RQ3 Can CFR achieve concurrent repair by integrating multiple repair front-ends?

RQ4 What is the impact on CFR performance with multiple-test executions?

4.5.1 Evaluation Setup

In our evaluation, we selected repair tools that employs an optimization technique for recompilation to boost repair performance. **Prophet**[79] is a learning-based technique that use a correctness model to prioritize and rank the candidate patches, which also employs aggregated compilation to minimize the cost of recompilation. **Darjeeling** implements statement-based transformation based on those introduced by GenProg[68] and optimizing the repair process using parallel computation via containerization. In our experiments, we set the search algorithm of Darjeeling to enumerate over all transformation patches within the search space instead of the traditional genetic search in GenProg[68].

We use the VulnLoc[122] benchmark which consist of a buggy program with a failing test-case exposing a security vulnerability. Since passing test-cases are not available, the performance on this benchmark is highly dependent on the recompilation cost. We also evaluate using ManyBugs[46] benchmark which consist of a test-suite for each defect that provides insights on the effectiveness of CFR for test validation. For our experiments we use the subset of ManyBugs benchmark as defined in [101]. Table 4.1 details the subjects we use in our evaluation.

Table 4.1: Experiment subjects and their details

Benchmark	**Program**	**Description**	**LOC**
VulnLoc	BinUtils	GNU Binary Utilities	2.7M
	CoreUtils	GNU Core Utilities	63K
	FFMpeg	Media Library	0.9M
	Jasper	Library for Images	28K
	LibArchive	Library for Archives	0.1M
	LibJPEG	Library for JPEG files	42K
	LibMing	Library for flash files	66K
	LibTIFF	Library for TIFF files	66K
	LibXML2	XML C Parser	0.2M
	Potrace	Tool for tracing	9K
	ZzipLib	Library for Archives	8K
ManyBugs	LibTIFF	Library for TIFF files	66K
	GZip	Library for Zip files	491K
	GMP	Math Library	145K

In our evaluation we collect several performance metrics to compare the effectiveness of using compilation free repair with existing state of the art repair tools. We define “latency” as the time duration to identify the first plausible patch since the start of the repair process. Additionally, we measure the throughput of each tool via “rate” which we define as the number of candidate patches enumerated per

Table 4.2: Experiment results for comparative analysis of CFR vs recompilation techniques

Tool	**Subject**	**Recompilation**			**Compilation-Free**			**Improvement**		
		latency	**candidates**	**rate**	**latency**	**candidates**	**rate**	**space**	**throughput**	**latency**
Prophet	BinUtils	625.00	2967	0.94	12.99	2968	1.69	1.00	1.81	48.11
	CoreUtils	64.00	14265	3.96	23.00	33455	8.14	2.35	2.05	2.78
	FFMpeg	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
	Jasper	167.00	21626	6.54	145.28	94135	26.05	4.35	3.99	N/A
	LibArchive	N/A	12969	3.60	6.16	68015	18.86	5.24	5.24	55.01
	LibJPEG	450.00	32964	9.15	8.18	57339	15.76	1.74	1.72	4.36
	LibMing	349.00	3145	0.87	79.99	24962	6.87	7.94	7.86	4.42
	LibTIFF	1126.71	55168	15.50	254.78	345067	95.11	6.25	6.14	N/A
	LibXML2	N/A	21880	6.08	50.42	61083	16.93	2.79	2.79	N/A
	Potrace	N/A	56622	15.72	62.04	201672	55.73	3.56	3.54	N/A
	ZzipLib	N/A	5969	5.23	271.31	115293	31.97	19.32	6.12	N/A
Darjeeling	BinUtils	N/A	123	0.04	13.97	900	1.26	7.32	28.13	N/A
	CoreUtils	2210.08	74	0.02	164.14	439	1.11	5.93	58.03	13.47
	FFMpeg	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
	Jasper	57.48	411	0.15	25.89	462	1.42	1.12	9.22	2.22
	LibArchive	78.03	217	0.06	45.45	709	0.74	3.27	13.24	1.72
	LibJPEG	107.18	694	0.18	24.89	853	1.04	1.23	5.83	4.31
	LibMing	73.39	77	0.05	46.36	613	0.76	7.96	16.29	1.58
	LibTIFF	562.38	360	0.09	36.44	1493	0.98	4.15	10.43	15.43
	LibXML2	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
	Potrace	124.00	682.00	403.00	49.84	682	0.74	1.00	3.96	2.49
	ZzipLib	203.43	403.00	0.13	48.62	912	1.27	2.26	9.75	4.18

second. All tools are configured to run in full-exploration mode; which will continue to generate patches even after finding one plausible patch until the timeout or the completion of exploring the search space; a plausible patch is one which passes all given tests. All our experiments were conducted using Docker containers on top of AWS (Amazon Web Services) EC2 instances. We used the c5a.8xlarge instance type for RQ1 and RQ2, which provides 32 vCPU processing power and 64GiB memory capacity. For RQ3 we used the c5a.24xlarge instance type, which provides 96 vCPU processing power and 192GiB memory capacity.

4.5.2 Comparison with Recompilation (RQ1)

We evaluate the effectiveness of CFR against existing recompilation techniques implemented in program repair. For our evaluation we compare against aggregated recompilation (i.e. Prophet) and single patch recompilation (i.e. Darjeeling). Prophet aggregates patches by selecting a single patch constructed for different patch locations, constructing a super-mutant and enabling the patch during runtime. In this evaluation for Darjeeling we used only a single thread since our comparison is on the performance difference between recompilation and compilation-free validation.

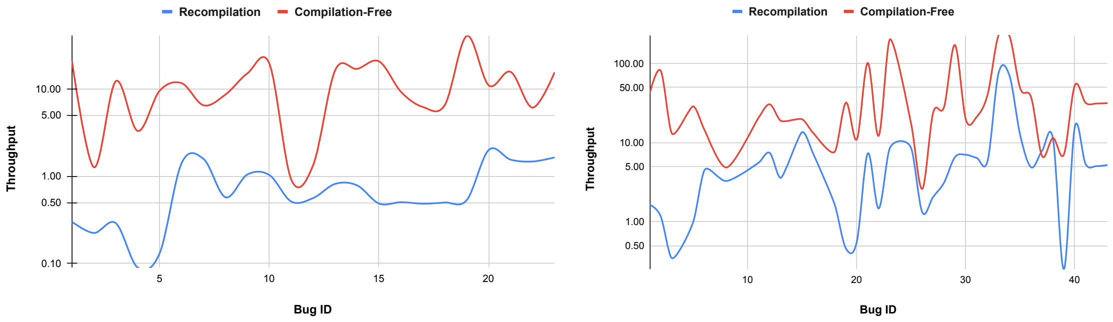
Results of our evaluation is summarized in Table 4.2. Columns "latency", "candidates", "ratio" and "rate" represents the average value for each subject in the benchmark for the specified tool. Column "Improvement" depicts the average performance gain for each tool on each subject, defined as x/y where x is the value with compilation free validation and y without it. CFR provides a significant performance gain over single patch recompilation. Since there is a compilation time delay between consecutive patch validations, a compilation-free technique provides better performance. On average single patch recompilation setup in Darjeeling explores 6 patches per minute while compilation-free enabled Darjeeling can explore 62 candidate patches per minute, which is an speedup of 10.3x. In terms of time to find the first plausible patch, single patch recompilation mode on average find a plausible patch in 427s while compilation-free mode finds a plausible patch in 55s which an improvement of 7.7x. In contrast to single patch recompilation, aggregated recompilation performs better since the recompilation cost per patch is reduced by distributing over multiple candidate patches. Prophet explores 425 candidate templates per minute, with compilation-free validation it explores 1364 patch templates per minute, which is an speedup of 3.2x. In terms of time to find a plausible patch, Prophet in recompilation mode find a plausible patch in 463s while compilation-free mode finds a plausible patch in 87s which an improvement of 5.3x. CFR provides performance gain over selective aggregated recompilation because aggregated recompilation there is a cost in latency since validation can be done only after the selecting a list of candidates patches from search space enumeration. Whereas, CFR can validate a patch concurrently as the enumeration progress. Although aggregated compilation has high throughput it is specialized for a specific tool, while a compilation-free approach can be generic to any repair tool.

RQ1 – Comparison with Recompilation: Compilation-Free Repair can achieve maximum speedup of 10.3x for throughput and 7.7x for latency in comparison to sequential recompilation.

4.5.3 Parallel Validation (RQ2)

In this experiment, we evaluate the overall end-to-end performance improvement by analyzing the throughput of each repair tool with different configurations. We evaluated each tool with recompilation and compilation-free validation. For CFR mode

Figure 4.4: Throughput of Darjeeling and Prophet



(a) Darjeeling

(b) Prophet

for each tool we employed sequential validation and parallel validation. Figure 4.4 presents the summarized results where each sub-figure depicts the throughput of each tool for different configurations. The horizontal axis presents the bugs and the vertical axis presents the throughput in patch per minute. Figure 4.4a presents the results for *Darjeeling*, where recompilation mode was configured *Darjeeling* to utilize 32 threads and Figure 4.4b presents the results for *Prophet*.

Darjeeling on average can validate 6 patches per minute using a single thread whereas using 32 threads improves the throughput to 49 patches per minute which is an improvement of 8.1x. Although parallelization using containers does provide performance improvement for throughput by over an order of magnitude, it does not improve better than compilation-free repair in sequential validation. In-fact, compilation-free repair with sequential validation provides similar results (i.e. 10.4x speedup) with less resource usage. Combining parallelization with compilation-free validation it can speedup to 825x times than a single thread performance. Prophet implementation of selective aggregated compilation performs better than Darjeeling since it enables to validate multiple patches in a single recompilation. Prophet enumerates 6 patch templates per second which results in an average latency of 463 seconds, with compilation-free parallelized validation it can validate 54 templates which is an improvement of 8x.

RQ2 – Parallel Validation: Compilation-Free Repair with Parallelism can achieve maximum speedup of 825.69x for throughput and 17.41x for latency in comparison to sequential recompilation.

4.5.4 Concurrent Repair (RQ3)

In this experiment we evaluate the effectiveness of concurrent repair of multiple repair front-end integration. For this purpose, we integrate several repair tools into one framework that concurrently explores its own search-space generating candidate patches that will be evaluated using compilation-free validation back-end. The framework is implemented using the producer/consumer design pattern in concurrent systems as shown in Figure 4.3. The repair tools will act as a front-end/producer that generates candidate patches as it explores the search-space defined by its enumeration strategies. A consumer that implements compilation-free validation will take each generated patch and validate concurrently. For this experiment we use 200 validation consumers while using one producer for each repair tool with total of 3 producers. Table 4.3 summarize the results of our experiments. Columns "t_p", "Candidates", and "Rate" represents the average value for each subject in the benchmark. Column "t_p" reports the average time taken to find a plausible patch, while the column "Rate" reports the average number of candidate patches validated per second.

Table 4.3: Experiment results with multiple repair integration

Subject	t_p	**Candidates**	**Rate**
Binutils	37.99	58609	17.97
Coreutils	25.32	127504	40.37
FFMpeg	N/A	N/A	N/A
Jasper	13.18	53728	30.07
LibArchive	9.65	71145	19.31
LibJPEG	14.16	113818	31.88
LibMing	20.15	84390	31.87
LibTiff	73.85	114248	36.31
LibXML2	52.54	93213	25.59
Potrace	6.30	88162	24.29
ZZipLib	9.73	34821	39.10

The integrated solution was able to repair most of the defects in the benchmark, for which fixes originated from different repair tools. The defects in FFMpeg subject was not repaired by any of the tool since the bug was not reproduced by the provided failing test-case. The average time to find a plausible patch is **26.9s** with an average throughput of 29.68 patches per second. A combination of the strength of different tools provides a coherent search for the defect, that can quickly identify and present a plausible patch to the developer for further analysis. A lightweight ranking based on the coverage information shows the top-10 ranked patches for some defects comprise of patches generated from different tools. Furthermore, the ranking is continuously updated as and when a new patch is generated, providing the capability of real-time ranking of plausible patches. To the best of our knowledge, this is the first repair system that integrates multiple repair front-ends to achieve concurrent repair, whereby multiple repair tool simultaneously search over different search spaces.

RQ3 – Concurrent Repair: Concurrent repair with multiple repair front-end integration can be achieved using a unified validation framework via compilation-free repair.

4.5.5 Impact of Multiple Tests (RQ4)

In this experiment, we evaluate the impact of having multiple test-cases on the effectiveness of compiler-free validation. We evaluated each tool with recompilation and compilation-free validation. For CFR mode for each tool we employed parallel validation. Table 5.3 presents the summarized results the throughput of each tool in terms of patches validated per second for different configurations. Results for *Darjeeling* is generated using 32 threads. For this evaluation we make use of a subset of the the popular test-suite repair benchmark ManyBugs [46]. In addition to the criteria specified in recent evaluations [101], we further filter the subjects for which the test-suite can be executed in parallel.

Table 4.4: Experiment results with multiple test-cases

Tool	**Subject**	**Throughput**		
		Original	**CFR**	**Speedup**
Prophet	LibTIFF	11.5	63.37	5.5
	GZip	3.37	21.42	6.35
	GMP	2.14	17.66	8.25
Darjeeling	LibTIFF	0.39	7.71	19.76
	GZip	1.05	18.56	17.67
	GMP	N/A	N/A	N/A

Patch validation with multiple test-cases impacts both recompilation and nonrecompilation performance. However, compilation-free repair is still able to achieve performance gain over recompilation for Prophet and Darjeeling with an average speedup of 6.7x and 18.7x respectively.

RQ4 – Impact of Multiple Tests: Multiple test-cases degrades overall throughput of repair, however compilation-free validation still provides significant performance gains.

4.6 Threats to Validity

There are several threats to validity of our approach, related to the external tools we use and the datasets that we use in our experiments. In our evaluation we do not cover all repair tools available, instead we cover the main recompilation strategies employed to minimize the validation cost: parallel compilation, sequential compilation and aggregated compilation. We seek to mitigate dataset bias by conducting our experiments on a wide variety of subjects including those studied by previous work. We chose VulnLoc [122] and ManyBugs [46] benchmarks which are well-known collection of defects in open-source projects. Using VulnLoc benchmark which only include one failing test-case provides insight into security vulnerability repair and ManyBugs benchmark provides insight into test-suite based repair. In addition, our experimental results explore the capability of repair tools to produce patches within a 1-hour timeout in contrast to related work. We use a 1-hours timeout based on the recent study which indicates developers prefer a maximum of 1 hour for automated repair tools [101].

On Search space representations We would also like to clarify that the contributions of this work are in speeding up search-based repair tools, which proceed on an explicit representation of the search space. Accordingly we have chosen two search-based repair frameworks: Prophet and Darjeeling. Prophet uses enumerative search to navigate a search space of patch candidates (along with use of machine learning to rank patch candidates which is not important for our purposes). Darjeeling is a search based repair framework produced by the GenProg research team which can plug and play different repair tools, notably GenProg which can use either enumerative or meta-heurisitic search. The Darjeeling implementation we worked with is based on the same explicit search space of patches and mutation operators as GenProg. Both Prophet and Darjeeling, we have shown significant performance improvements. However, apart from explicit generate-and-validate repair tools which work on explicitly represented search spaces, there are other repair tools which work on implicit representations of the search space of patches. This includes F1X [89] which uses value/dependence-based equivalence relations to represent the patch space, or SemFix [100], Angelix [92] which use symbolic representations of the search space. We make no claims of performance improvement (or performance reduction) with respect to these tools which work on an implicit representation of the search space. To study the performance issues with respect to such tools would need a study of several factors, including (a) whether the implicit representation of the patch space is built at once or on-the-fly, (b) whether the implicit search space representation construction is dependent on run-time artifacts (hence necessitating compilation) or compile-time artifacts, and so on. Understanding these issues can be a topic of future work; they have not been investigated in this paper. Our performance improvement claims and results only pertain to generate and validate program repair tools working on explicit search space representations.

4.7 Summary

Patch validation is one of the steps in program repair specifically for generate and validate techniques whereby candidate patches are validated using a suitable test oracle. A key step for validation is recompilation which translates the candidate patch into an executable binary that can be tested against a test oracle. Incremental recompilation per patch is low in cost, however accumulated over thousands of candidate patches the cost is significant especially for program repair under time constraint. In this chapter, we look at a compilation-free validation technique that replace the compiler from the repair-loop with a lightweight interpreter that is able to achieve the same result at a low cost. Using on-the-fly patch validation with existing program repair tools for C/C++ programs we show the benefits of compilation-free repair. We show the benefit of a concurrent repair framework that integrates multiple repair tools with a single validation back-end to improve throughput and latency in program repair. The experimental results using Prophet and Darjeeling show that CFR can improve the overall performance of APR using such an integration to work concurrently which can simultaneously explore multiple search spaces thus enabling repair to access a larger search-space by over an order of magnitude compared to existing repair tools.

Chapter 5

Vulnerability Repair using a Program-Specification

Automated program repair reduces the manual effort in fixing program errors. However, existing repair techniques modify a buggy program such that it passes given tests. Such repair techniques do not perform well for specific program errors such as security vulnerabilities which does not provide a large-enough test-suite to infer a specification for the program repair. Hence, existing state-of-the-art test-driven repair techniques generate a large set of plausible patches that are overfitting the provided test-cases and do not mitigate the security vulnerability entirely. We propose a novel approach that takes into account a user-provided specification that can guide the repair process while reasoning the semantic correctness of the generated patches with respect to the provided program specification. This chapter introduces **concolic program repair** for integrating a user-provided specification to guide the repair process and shows how it can improve the state-of-the-art repair tools to generate high-quality repair for security vulnerabilities. The chapter starts with an overview of our approach on how repair can be guided to fix security vulnerabilities using a program-specification. It continues with an illustrative example that shows the advantages of concolic program repair and afterwards provides a discussion of the technical details. The chapter concludes with a preliminary evaluation of the effectiveness of our novel approach in repairing reported security vulnerabilities in large-scale real-world software applications.

5.1 Overview

In this chapter, we reflect on the problem of *patch overfitting* [110, 133, 80] in program repair, in order to produce high-quality patches for security vulnerabilities which does not provide sufficient test-cases to mitigate the problem of *patch overfitting.* To reduce exposure of software security vulnerabilities, it is essential to generate patches quickly, especially for time-critical software vulnerabilities. Automated program repair can generate patches quickly but does not provide correctness guarantees, which is of essence to remedy exploits of the security vulnerability. Our goal is to devise a repair technique that is of any-time patching algorithm; the algorithm can be stopped at any time. However, the longer it is run, the greater is the coverage of the input space, and the greater is our confidence that the patch produced works for a large class of test inputs. To ensure coverage of the test input space, we use concolic path exploration for **automated test generation**.

We use concolic execution [42] to generate test inputs, and additionally to generate constraints for the patch refinement, to make them work for those test inputs. We leverage a user-provided specification to detect incorrect behavior for the generated test inputs. Such specification does not need to be a full specification with regard to the program's correctness. Partial specifications like an assertion at a specific location, or the absence of crashes in a specific location, can be already sufficient to detect overfitting patches. Our outlook is to use concolic execution for computing path constraints and patch constraints at the same time. By making the symbolic execution technology serve such a dual purpose, we can systematically traverse a large portion of the test input space, and find out patch patterns which work for those traversed test inputs. Given a longer time budget, we obtain greater path coverage, and rule out a large number of patch candidates, thereby reducing overfitting in program repair.

Realizing such a dual-purpose usage of symbolic execution, requires us to overcome many technical challenges. First, our symbolic execution engine needs to compute path constraints containing both input variables and patch variables. Though the patch variables are higher order variables, we avoid developing a second order symbolic execution engine for scalability reasons. Instead we provide a first order encoding of path constraints and patch constraints which contain (first order) input variables along with certain additional parameters to succinctly represent sets of patches. Secondly, and more importantly, there are additional sources of path infeasibility as compared to traditional concolic/symbolic execution, in our setup. In traditional concolic execution, a path is deemed infeasible if the path constraint is unsatisfiable. In our setup, the path contains a hole for the patch location, and we maintain a pool of patch candidates which diminishes as more paths are explored. Hence if none of the remaining patch candidates can be inserted into the patch location, we also deem the path as infeasible.

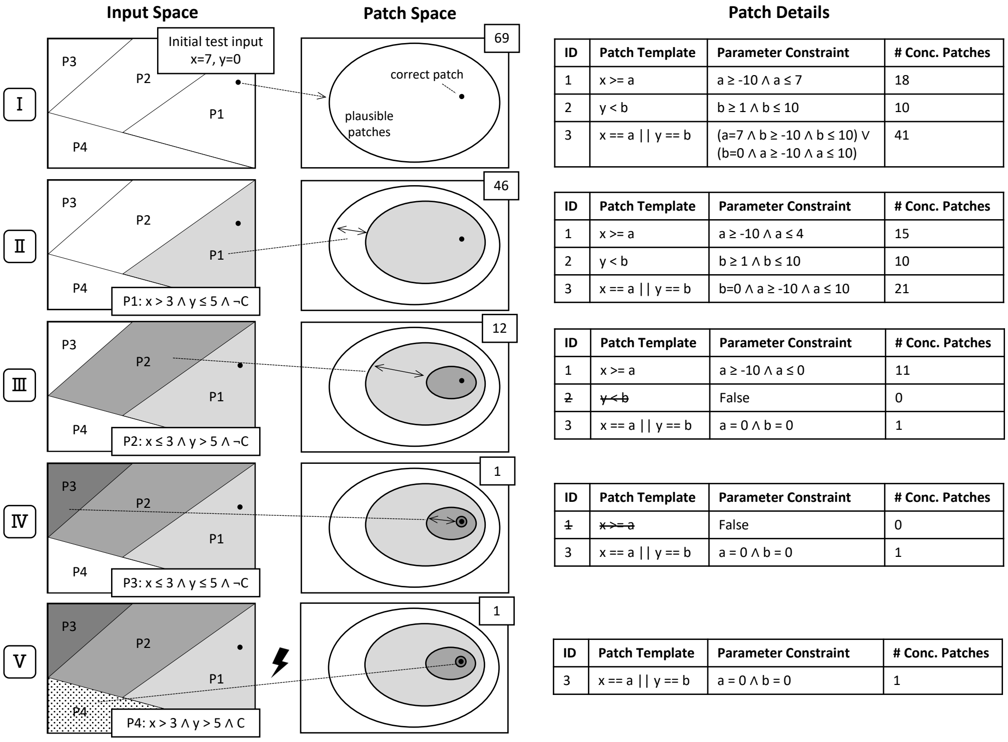
The benefits of our concolic approach for patch generation are shown by the experimental evaluation of its efficacy in repairing a large set of security vulnerabilities curated in recent works [40] based on Google's OSS-Fuzz infrastructure. The tool embodying our concolic program repair approach is called CPR, an abbreviation indicating the resuscitation of programs via appropriate fixes.[1]

We propose the path exploration in concolic execution as a mechanism to traverse the program input space and patch space simultaneously. The main insight is that simultaneous exploration of input and patch space helps tackle patch overfitting, which is a key problem in the area of automated program repair [110, 133] and a necessity to generate quality-patches for security vulnerabilities. Our repair tool CPR generates correct patches for a variety of specifications or oracles including crash-freedom (absence of observable vulnerabilities), and satisfaction of assertions — as shown by our experiments.

5.2 Example

In this section we show the advantages of *concolic program repair* by illustrating its usage for the repair of a security vulnerability in a real-world application. We make use of the security vulnerability reported as CVE-2016-3623 discovered in the LibTIFF library v4.0.6 (see Listing 5.2). LibTIFF is a popular open-source library that provides support for the Tag Image File Format (TIFF), a widely used format for storing image data. CVE-2016-3623 represents a divide-by-zero vulnerability, which allows a remote attacker to cause a denial of service by setting malicious inputs to the program `rgb2ycbcr`. Listing 5.2 depicts the relevant code

[1] Resuscitating a program, like what Cardio-pulmonary Resuscitation (CPR) does to a patient.



ID	Patch Template	Parameter Constraint	# Conc. Patches
1	x >= a	a ≥ -10 ∧ a ≤ 7	18
2	y < b	b ≥ 1 ∧ b ≤ 10	10
3	x == a || y == b	(a=7 ∧ b ≥ -10 ∧ b ≤ 10) ∨ (b=0 ∧ a ≥ -10 ∧ a ≤ 10)	41

ID	Patch Template	Parameter Constraint	# Conc. Patches
1	x >= a	a ≥ -10 ∧ a ≤ 4	15
2	y < b	b ≥ 1 ∧ b ≤ 10	10
3	x == a || y == b	b=0 ∧ a ≥ -10 ∧ a ≤ 10	21

ID	Patch Template	Parameter Constraint	# Conc. Patches
1	x >= a	a ≥ -10 ∧ a ≤ 0	11
~~2~~	~~y < b~~	False	0
3	x == a || y == b	a = 0 ∧ b = 0	1

ID	Patch Template	Parameter Constraint	# Conc. Patches
~~1~~	~~x >= a~~	False	0
3	x == a || y == b	a = 0 ∧ b = 0	1

ID	Patch Template	Parameter Constraint	# Conc. Patches
3	x == a || y == b	a = 0 ∧ b = 0	1

x = horizSubSampling, **y** = vertSubSampling, **C** = CONDITION
Illustrative concolic exploration for example CVE-2016-3623 in Listing 5.2 as the simultaneous exploration of the input space and the patch space. The rows I, II, III, IV, and V represent multiple exploration steps. The columns show the increasingly covered *Input Space*, the decreasing *Patch Space*, as well as more details on the identified patches. The patch space is in general limited by the synthesis language (denoted by the rectangular around the patch space illustration). The number on the top right of the patch space illustration denotes the total number of concrete patches included in this patch space.

Figure 5.1: Illustration of Concolic Program Repair

```
........
static int
cvtRaster(TIFF* tif, uint32* raster, uint32 width,
    uint32 height)
{
    uint32 y;
    tstrip_t strip = 0;
    tsize_t cc, acc;
    unsigned char* buf;
    uint32 rwidth = roundup(width, horizSubSampling);
    uint32 rheight = roundup(height, vertSubSampling);
    uint32 nrows = (rowsperstrip > rheight ?
        rheight : rowsperstrip);
    uint32 rnrows = roundup(nrows,vertSubSampling);
    if (CONDITION) return 0;
     /* potential divide-by-zero error */
    cc = rnrows*rwidth + 2 * ((rnrows*rwidth)
        / (horizSubSampling*vertSubSampling));
    ........
}
```

Figure 5.2: CVE-2016-3623: Divide by Zero in LibTIFF v4.0.6

snippet, which could lead to a divide-by-zero error at line 263 if the two variables `horizSubSampling` and `vertSubSampling` are not sanitized for invalid inputs. We have added a fix template in line 261, where the condition can be generated using most state-of-the-art repair tools.

Repair process. Concolic program repair works on a high-level in three phases: (1) patch pool construction, (2) path exploration, and (3) patch reduction. The phases (2) and (3) are performed in an alternating manner: The path exploration provides input partitions (in form of path constraints), and the patch reduction refines abstract patches and rules out patches that fail the user-provided specification for the current input partition.

Illustration. Figure 5.1 illustrates the simultaneous space reduction (i.e., the interplay between path exploration and patch reduction): as we explore the input space, we are able to narrow down and refine the patch space (steps I, II, III, and IV), while at the same time we leverage the patch space to skip parts of the input space, which are not feasible with the available patches (step V). Therefore, each row I, II, III, IV, and V in Figure 5.1 represents an exploration step, which represents an increase of the *input space* coverage and a potential reduction of the *patch space*. The input space for this example is partitioned into 4 compartments *P1*, *P2*, *P3*, and *P4*, which are defined by the corresponding path constraints. Note that the constraints in Figure 5.1 show only the relevant parts for this example and further assume a control location, which compares the relevant variables `horizSubSampling` and `vertSubSampling` with the given constants. These path constraints are chosen artificially for this example (since details of `roundup` are not shown). As mentioned in Figure 5.1, we refer to `horizSubSampling` and `vertSubSampling` as x and y respectively as a notational short-hand. Our patch space is generally limited by the synthesis language (denoted by the rectangle around the patch space illustration). In order to illustrate the overall reduction in terms of concrete patches, the box in the top right corner of the patch space shows the total number of concrete patches included in this patch space. Please note that Figure 5.1 does not show the exploration of all possible input partitions, and hence, shows only a part of the input exploration for illustration purposes.

Patch pool construction. In this example, our approach starts with synthesizing a set of plausible patches based on an initial test case with `x=7`, `y=0` (see step I in Figure 5.1). We assume that the user-defined *specification* states that there should be no divide-by-zero error at line 263 in Listing 5.2, i.e., that $x * y \neq 0$. The set of plausible patches is shown as the oval in the *Patch Space* column. Note that we assume that the correct patch is included in this set. The table on the right side of Figure 5.1 shows an illustrative list of patch templates (aka abstract patches) generated by our synthesizer. As abstract patches we consider boolean and integer expressions, which include program variables (e.g., x and y) and parameters (e.g., a and b). During the repair process, the parameter values are captured by a certain constraint (see column *Parameter Constraint*), which covers a set of concrete patches and limits the search space. The column *# Concr. Patches* show how many concrete patches are covered by the corresponding abstract patch. For this illustrative example, we assume that the parameter values are initially in the range [-10, 10]. The constraints shown in the table are already modified by the synthesizer to pass the initial test case. In the following paragraphs, we will provide more detailed information on the interplay between path exploration and patch reduction.

Input partition P1 for patch 1. Starting with the initial input, concolic execution provides us with the input partition P1 (defined by the corresponding path constraint). Step II in Figure 5.1 represents the first repair iteration. For every abstract patch, we check whether a violation of the specification is feasible with the current path constraint. If yes, we try to refine the constraint on the parameter values. The light-grey shaded area in the patch space indicates the refinement to the patch space as we explore the respective path of P1. In order to refine patch 1, we search for models of:

$$\underbrace{x > 3 \wedge y \leq 5 \wedge \neg(x \geq a) \wedge a \in [-10, 7]}_{\substack{\text{path constraint P1 complemented with} \\ \text{patch 1}}} \;\wedge\; \underbrace{(x * y = 0)}_{\substack{\text{condition for} \\ \text{specification} \\ \text{violation}}}$$

Every satisfying assignment reveals a possibility to violate the specification with the current path constraint and patch 1. In order to make this formula unsatisfiable, we need to remove the values $\{5, 6, 7\}$ from the constraint on a. Therefore, the refined variant of patch 1 is: $x \geq a$ with $a \in [-10, 4]$ (see table on the right side of row II in Figure 5.1). This refinement removes 3 concrete patches from the patch space.

Input partition P1 for patch 2. In order to test patch 2 on the input partition P1, we again first check whether it is possible to violate the specification with the current path constraint and patch 2. The formula to test would be: $x > 3 \wedge y \leq 5 \wedge \neg(y < b) \wedge b \in [1, 10] \wedge (x * y = 0)$. However, this formula is unsatisfiable, and hence, patch 2 cannot be refined in this step.

Input partition P1 for patch 3. For patch 3 we need to test: $x > 3 \wedge y \leq 5 \wedge \neg(x = a \vee y = b) \wedge (a = 7 \wedge b \in [-10, 10] \vee b = 0 \wedge a \in [-10, 10]) \wedge (x * y = 0)$. For this formula, only $y = 0$ is the feasible condition for a violation. Therefore, all parameter value combinations, for which $b \neq 0$ are models for a specification violation and need to be removed from the parameter constraint during refinement. The resulting parameter constraint is: $(a = 7 \wedge b \in [0] \vee b = 0 \wedge a \in [-10, 10])$, which can be simplified to $b = 0 \wedge a \in [-10, 10]$.

Exploration of P2 and P3. In order to generate a new input, the current path constraint of P1 can be mutated, e.g., by flipping constraints in P1 (as in concolic execution), and solved with an SMT solver. For example, we could retrieve the input `x=0`, `y=6` corresponding to the path constraint P2: $x \leq 3 \wedge y > 5 \wedge \neg C$ (see step III in Figure 5.1). While exploring P2, the parameter constraint in patch 1 can be refined to $a \in [-10, 0]$. Patch 2 does violate the specification for P2 for *all* available parameter values. Therefore, patch 2 cannot be refined and needs to be removed in step III. Finally, the parameter constraint in patch 3 can be refined to $a = 0 \wedge b = 0$, i.e., there is only one concrete mapping left for this patch. In fact, patch 3 now is semantically equivalent to the *correct* patch. Step IV in Figure 5.1 shows one final step, where patch 1 can be removed and patch 3 remains as the correct patch.

Non-Exploration of P4. Step V in Figure 5.1 shows the consideration of P4 with the path constraint $x > 3 \wedge y > 5 \wedge C$. One of our key ingredients is, when generating a new input, we ensure the feasibility of the corresponding path constraint by selecting an appropriate patch from our patch pool. The above mentioned path constraint for P4 is satisfiable; however, our approach would not explore it because there is no patch in the current patch pool, which would allow taking this path.

Advantages of concolic program repair. Our approach has the major advantage to explore both spaces, input and patch, *simultaneously*, saving a significant cost in terms of time and space enumeration: (1) we refine the patch space based on the exploration in the input space, while (2) we also can rule out parts of the input space, which contradicts with the patch space. We are able to reason about a *large portion* of concrete patches with every single iteration of concolic execution by using *abstractions* in the *patch space*. For example, with three repair steps (II, III, and IV) we can reduce the patch space by 68 concrete patches. In general, the more paths we explore, the better the refinement would be, thus finding the most accurate patch. Furthermore, instead of focusing only on specific inputs but rather on the obtained path constraint, we are able to test a *large portion* of the *input space* captured by an input partition. Additionally, as illustrated in our example, our approach performs some *path reduction*: during concolic exploration, we make sure that for every new generated input, there is at least one patch in the current patch pool, which can exercise the corresponding path. Otherwise, the path will not be explored.

In conclusion, these advantages allow us to reduce the pool of candidate expressions, as compared to existing state-of-the-art techniques like counterexample-guided inductive synthesis (CEGIS) [134, 135] and ExtractFix [40].

5.3 Methodology

Our proposed workflow *concolic program repair* incrementally explores the input space, while refining the patch space. The workflow involves symbolic execution, patch synthesis and concolic exploration. In this section, we discuss each phase in more detail.

5.3.1 Patch Definition

Our technique supports two notions of patches: *concrete* and *abstract*. An abstract patch represents a patch template, which contains parameters that can have values satisfying a specified constraint. Concrete patches do not include such parameters. Our methodology focuses on abstract patches because, having abstract patches, the repair process needs to generate and maintain a smaller amount of patch candidates. Furthermore, the patch space reduction can attempt to refine the parameter constraints before discarding a patch. Therefore, we define a patch ρ as the 3-tuple

$$(\theta_\rho, T_\rho, \psi_\rho)$$

with the set of program variables X_P, the corresponding subset of input variables $X \subseteq X_P$, and the set of template parameters A:

- $\theta_\rho(X_P, A)$ denotes the repaired (boolean or integer) expression
- $T_\rho(A)$ represents the conjunction of constraints $\tau_\rho(a_i)$ on the parameters $a_i \in A$ included in θ_ρ:
$$T_\rho(A) = \bigwedge_{a_i \in A} \tau_\rho(a_i)$$
- $\psi_\rho(X, A)$ is the *patch formula* induced by inserting the expression θ_ρ into the buggy program

This patch definition covers both notions *abstract* and *concrete*. For concrete patches the set of parameters A is either empty and T_ρ is trivially `True`, or the constraints on the parameters $a_i \in A$ allow only one concrete value each.

Example. Assuming a buggy location in a program like **if**(ρ)**then..else..**, where the patch ρ is included in the if condition. Then a repaired expression could be $\theta_\rho := x > a$ with the parameter value constraint $T_\rho = \tau_\rho(a) := (a \geq -10 \wedge a \leq 10)$ and the corresponding patch formula $\psi_\rho := x > a$.

Patch Formula. In our notation ψ_ρ does not represent the patch expression but rather the constraint induced by the patch. For our approach a patch is technically represented as an expression tree, which can be transformed into an SMT formula, by considering the semantics of the operators (or components) appearing in the expression θ_ρ. The information about the patch location (i.e., where the repaired expression will be inserted) and the transformed expression tree is what we call the *patch formula.* Therefore, if the patch represents the right hand-side of an assignment like y=ρ with $\theta_\rho := x - a$, then the patch formula is derived as $\psi_\rho := y = x - a$, using the patch context information. We acknowledge that such a patch formula is generally not required for the definition of a patch. In fact, the patch formula can be derived from combining the information about the patch location and the patch expression (see Section 5.3.2.3). However, our approach technically requires such an artifact in order to reason about the patch.

5.3.2 Concolic Repair Algorithm

As *input*, our approach requires the buggy program, a repair budget, the fault locations, a user specification, the language components for the synthesis, and optionally, a set of initial test cases. The user specification identifies a constraint on the desired program behavior (in addition to satisfying the given test cases). It does not need to be a complete formal specification of the correct program behavior, but represents a constraint on the expected observation, provided as a logical formula. For example the user can assert crash-freedom or some specific logical behavior (e.g., a constraint on the resulting output). If no error-exposing input is available, we need to generate at least one failing input (with regard to the user-provided specification) to start the concolic exploration. Therefore, we can use offline techniques like Directed Greybox Fuzzing [13]. Note that the generation of the one failing test is a pre-processing to our technique. Otherwise, we assume that at least one failing test is available, which our method seeks to repair, apart from making sure that the user-provided specification holds for all paths traversed via concolic exploration.

As *output* our approach produces a set of patches, which satisfy the initial test case (repairing the given failing test case, if one is available) and which do not violate the given specification for (a subset of, depending on the repair budget) the other paths of the program. The patches are *ranked* based on the evidence we see during input space exploration.

Algorithm 1 shows the general workflow of concolic repair, which implements three phases: (1) *patch pool construction* (see Section 5.3.2.1), (2) *path exploration*

Algorithm 1: General Concolic Repair

```
   Input: set of initial test cases $I$, buggy locations $L = (patchLoc, bugLoc)$,
          budget $b$, specification $\sigma$, language components $C$
   Output: set of ranked patches $P$
1  P ← Synthesize(C, I, L)
2  while $P \neq \emptyset$ and CheckBudget(b) do
3      $t$, $\rho$ ← PickNewInput($P$)
4      if no input $t$ available then
5          return P
6      end
7      $\phi_t$, $hit_{patch}$, $hit_{bug}$ ← ConcolicExec($t$, $\rho$, $L$)
8      if $hit_{patch}$ then
9          $P$ ← Reduce($P$, $\phi_t$, $\sigma$, $hit_{bug}$)
10     end
11 end
12 return P
```

(see Section 5.3.2.2), and (3) *patch reduction* (see Section 5.3.2.3). The initial phase of synthesis produces a pool of patches P (see line 1 in Algorithm 1) by leveraging a component-based synthesizer. This patch pool is going to be refined in the following repair loop (see line 2 to 11). The repair loop itself will be continued as long as there are remaining patches to refine or the repair budget is not exceeded. In phase (2) (i.e., inside the repair loop), we pick a new input t to explore more program paths (see line 3). With input t we also retrieve a patch candidate ρ from the patch pool P, such that inserting ρ in the patch location allows t to have a feasible path in the patched program. If there is no such input t available, then there is no more input space to explore and the algorithm will return the identified patches (see line 4 to 6). Otherwise, we perform a concolic execution of the program with input t, patch candidate ρ, and the information about the:

- *patch location*, where the **repair** is located and
- *bug location*, where the buggy behavior is **observable**.

It results in the path constraint ϕ_t and whether the patch location (hit_{patch}) and the bug location (hit_{bug}) have been exercised by the execution (see line 7). Afterwards, in phase (3), we aim to reduce the patch pool P based on the current observations and the given specification σ. Before calling the Reduce function in line 9, we check whether the current path actually exercises the patch location (see line 8), otherwise there is no reduction possible.

5.3.2.1 Patch Pool Construction

In order to generate the initial patch pool P we leverage a component-based synthesizer, which focuses on the synthesis of boolean and integer expressions. Our approach assumes that the necessary patch-ingredients are provided as input to our technique. This includes the available program variables and the arithmetic/comparison operators for the synthesis. Before starting the actual synthesis we employ a *controlled* symbolic execution [92] to retrieve the path constraints for the initial test cases. Therefore, we mark the patch variables as symbolic at the patch location. The result of this symbolic execution is a set of path constraints with their corresponding expected outputs given by the test cases.

The synthesis starts with generating a set of expression trees based on the available components and the required expression type at the patch location. We support the arithmetic operations $\{+, -, *, /\}$ as well as the remainder operation, the comparison operators $\{=, \neq, <, \leq, >, \geq\}$, the boolean operators $\{\wedge, \vee, \neg\}$, and usage of parameters like $\{a, b, c, ...\}$. More components can be easily added to our synthesizer by providing them in the SMT-LIB format. For example, for each program to be repaired, the available variables are provided as additional components to the synthesizer. The final set of expression trees contains all feasible combinations of the given components that fit the required expression type. Afterwards, the synthesizer enumerates over these trees and validates that the corresponding expressions repair the program for the constraints retrieved by the controlled symbolic execution. All successfully validated expression trees will be put in the resulting patch pool. If the expression tree includes parameters, the synthesizer will generate a constraint on these parameters (based on a pre-selected range).

5.3.2.2 Path Exploration

The path exploration is concerned with two issues: (a) how to pick a new input t and (2) how to efficiently retrieve the corresponding path constraint ϕ_t. In the first loop iteration the new input is chosen based on the provided test cases or randomly if there are no test cases available. Afterwards, based on the previous

Algorithm 2: Reduce function

```
Input: patch pool $P$, path constraint $\phi$, specification $\sigma$, bug location hit $hit_{bug}$
Output: reduced patch pool $P'$
$P' \leftarrow P$
for $\rho \in P$ do
    $\pi \leftarrow \phi(X) \wedge \psi_\rho(X, A) \wedge T_\rho(A)$
    if IsSat($\pi$) then
        if $hit_{bug}$ then
            $P' \leftarrow P' \setminus \rho$
            $T'_\rho \leftarrow$ RefinePatch($\phi$, $\rho$, $T_\rho$, $\sigma$)
            if $T'_\rho$ False then
                $P' \leftarrow P' \cup \{\rho$ with $T'_\rho\}$
            end
        end
        UpdateRanking($\rho$)
    end
end
return $P'$
```

path constraint, the PickNewInput function (see line 3 in Algorithm 1) applies generational search [43] to obtain new inputs: by negating every suffix term in the constraint, we can retrieve the maximum number of new path constraint prefixes.

While checking the satisfiability of the obtained path constraint prefixes, we also determine whether there exists a patch candidate ρ in our current patch pool, which allows to exercise this path. In this way, we prune paths, for which no patch is feasible. We call this pruning of the input space *path reduction*. After checking the satisfiability, we can generate a set of new inputs, which are ranked based on how often they trigger the execution of the patch and bug location. In this way, a set of new inputs is maintained, which can be worked on and extended in every repair iteration. The complete path constraint is then retrieved by concolically executing the new input, and injecting the patch formula ψ_ρ (for a patch expression ρ) into the path constraint.

5.3.2.3 Patch Reduction

The Reduce function in Algorithm 1 (see line 9) tries to shrink the patch pool and to possibly refine the available abstract patches. Its workflow is shown in Algorithm

2.

Criterion for Patch Reduction: For every patch ρ in the patch pool P we need to make sure that there is no violation of the specification σ for all inputs that are specified by the given path constraint. Otherwise, the patch needs to be removed. More specifically, we need to make sure that there exist parameter values parameters $a_i \in A$ within in the constraint $T_\rho(A)$ so that for all inputs $x_i \in X$, which satisfy the path constraint $\phi(X)$ and the patch formula $\psi_\rho(X, A)$, there is no violation of the specification $\sigma(X)$. Given $A = \{a_1, a_2, .., a_n\}$ and $X = \{x_1, x_2, .., x_m\}$, this means:

$$\exists a_1, a_2, .., a_n \forall x_1, x_2, .., x_m : \\ \phi(X) \wedge \psi_\rho(X, A) \wedge T_\rho(A) \implies \sigma(X) \quad (5.1)$$

In our approach we do not only ensure that there exists **one** value for each parameters a_i, but we iteratively refine the constraint $T_\rho(A)$ to reduce the patch space as much as possible and to ensure that the specification holds for **all** (refined) values for each parameter a_i:

$$\forall a_1, a_2, .., a_n \forall x_1, x_2, .., x_m : \\ \phi(X) \wedge \psi_\rho(X, A) \wedge T_\rho(A) \implies \sigma(X) \quad (5.2)$$

We want this formula (2) to hold after refinement, and hence it is used to guide our abstract patch refinement.

Reduction Algorithm: Algorithm 2 describes the reduction function for abstract patches. The function iterates over every patch and searches for specification violations. Before calling the patch refinement in line 7, there are two additional pre-checks, to make sure that we can reason about the patch within the current path constraint. First we check whether the path constraint ϕ and the current patch ρ (see line 3 and 4) are feasible. Secondly, we check whether the bug location is exercised by the current execution (see line 5) so that the buggy behavior is observable.

If both checks are passed, then we investigate whether the patch ρ with constraint T_ρ needs to be refined by searching for counterexamples for formula (2). The only option for the patch refinement, based on our definition of abstract patches (see Section 5.3.1), is to refine the constraint T_ρ. The implementation details for the patch refinement are presented in Section 5.4.3. If no refinement is feasible, then the patch will be eventually removed.

Patch Ranking: In addition to reducing the patch space, our approach attempts to rank the remaining patches. The rank of each patch ρ will be increased as long as the patch is feasible with the path constraint ϕ (see line 12 in Algorithm 2). Otherwise the ranking will be not modified because we cannot reason about the patch with regard to the current path constraint. If the path exercises the bug location, then the patch will be ranked additionally higher (as compared to the situation where it does not exercise the bug location). Intuitively, this means that (1) patches that are compatible with the current path constraint will be ranked higher because we have seen more evidence for their correctness (in terms of the explored input space). In addition, (2) patches that also exercise the bug location will be ranked even higher because they exercised the program location, where potential errors are observable. Patches that are compatible with the path constraint and do not exercise the bug location could still be erroneous, but there has been no possibility to observe the error. We only rank those patches which do not show any violation of the specification for the explored input space.

In addition, we deprioritize patches that change the program behavior significantly, specifically *deletion of functionality* — which can happen if the guard of a conditional statement is changed by a patch to tautologies or their negation. Based on our formula (2) we cannot remove these patches because they do not violate the specification. However, functionality deletion is in general not desirable; as stated in a recent study [110], this kind of functionality deleting patches are present in the earlier works on search-based program repair and are overfitting. Although we cannot remove these patches, our patch ranking mechanism deprioritizes them. Therefore, for all patch candidates, we check whether the insertion of the patch affects the control flow of the inputs flowing through the path (even if the insertion of the patch does not violate the user-provided specification). We deprioritize such patches, and increase the rank of the other patches, and this ranking fine-tuning is accumulated over all the paths explored. Further fine-tuning of this heuristic is possible via model counting [44, 21] to find the proportion of inputs in a path affected by a patch insertion.

5.4 Implementation

In this section we provide necessary implementation details for our tool CPR.This includes details about the generation of patch formula ψ as indicated in our patch definition (see Section 5.3.1), details on the ranking heuristics in our path exploration (see Section 5.3.2.2), and the algorithm for the patch *refinement* needed for the patch reduction (see Section 5.3.2.3).

5.4.1 Patch Formula Generation

Technically, a patch is represented as an expression tree including the expression components: the program variables, constants, the patch parameters, and arithmetic/comparison operators (depending on the used language components). Each expression tree can be transformed into an SMT formula, by considering the semantics of the operators (or components) appearing in the expression. For concolic execution, the program is instrumented with a symbol at the patch location, which represents a second-order variable because it depends on the input variables and the patch parameters. The information about the symbol and the transformed expression tree is what we call the *patch formula*. After concolic execution, we combine the current path constraint with the patch formula, which essentially replaces the symbol with the constraint derived from the patch. This results in an SMT formula, which can be solved with an off-the-shelf SMT solver in order to reason about patch with regard to the current path constraint.

5.4.2 Path Ranking

As mentioned in Section 5.3.2.2 the inputs (with their corresponding path constraints) are ranked based on how often the program exercises the patch and bug location. Technically, this can be determined by executing the program with the inputs, followed by the examination of the collected execution traces. However, this is rather expensive on an instruction level. Instead we examine the already available path constraint prefixes and count the number of occurrences of the corresponding symbols. Note that we instrument the patch location with the patch symbol (see Section 5.4.1), and similarly, we instrument the bug location with a symbol to inject the given specification as a constraint over the program variables. For paths with an equal count, we check in which path constraint the occurrences happen earlier, to break ties further.

5.4.3 Abstract Patch Refinement

During patch space reduction (see Algorithm 2) we try to refine the available abstract patches whenever we identify a corresponding violation of specification σ. This is achieved by efficiently refining the parameter constraint T_ρ of the abstract patch ρ as shown in Algorithm 3.

Removal of non-refinable constraints. Before starting the fine-grained refinement of T_ρ, the Algorithm 3 checks whether there is a refinement of T_ρ feasible, which will make the specification pass. It checks whether (a) the conjunction of the path constraint with the specification (see formula ω_{pass1} in line 1) is satisfiable, followed by the check whether (b) the conjunction of the path constraint with the current patch constraint still allows to pass the specification (see formula ω_{pass2} in line 3). If (a) is satisfiable, but (b) is unsatisfiable, the parameter constraint does not contain any value that repairs the specification violation, and hence, can be discarded completely.

Counterexample exploration. After these initial checks, the algorithm searches counterexamples for the general formula (2) from Section 5.3.2.3 (see formula ω_{fail} in line 8). They capture violations of the specification, which need to be excluded by our refinement of T_ρ. If there exists no such model for formula ω_{fail}, then the parameter constraint needs no further refinement and the current constraint can be returned (see line 31). But if there is a model m_A, the Split function removes the model from the current constraint T_ρ and splits it into multiple regions (see line 11).

Region representation. We assume that the parameter constraint can be split into k regions $R = \{r_1, r_2, ..., r_k\}$ so that the constraint represents the disjunction of the separate regions. This limits the search space during refinement and can lead to removal of regions, which do not satisfy the specification. For example, consider a parameter space with one parameter a and the constraint $T_\rho(a) := (l \leq a) \land (a \leq u)$.

Algorithm 3: RefinePatch function

```
Input: path constraint $\phi$, abstract patch $\rho$, parameter constraint $T_\rho$,
       specification $\sigma$
Output: refined constraint $T'_\rho$
$\omega_{pass1} \leftarrow \phi(X) \wedge \sigma(X)$
if IsSat($\omega_{pass1}$) then
    $\omega_{pass2} \leftarrow \phi(X) \wedge \psi_\rho(X, A) \wedge T_\rho(A) \wedge \sigma(X)$
    if $\neg$IsSat($\omega_{pass2}$) then
        return False
    end
end
$\omega_{fail} \leftarrow \phi(X) \wedge \psi_\rho(X, A) \wedge T_\rho(A) \wedge \neg\sigma(X)$
$m_A \leftarrow$ GetModel($\omega_{fail}$)
if $m$ exists then
    $R = \{r_1, r_2, .., r_k\} \leftarrow$ Split($T_\rho$, $m_A$)
    if $R = \emptyset$ then
        return False
    else
        $R' \leftarrow \{\}$
        for $r_i \in R$ do
            $\pi \leftarrow \phi(X) \wedge \psi_\rho(X, A) \wedge r_i(A)$
            if IsSat($\pi$) then
                $r'_i \leftarrow$ RefinePatch($\phi$, $\rho$, $r_i$, $\sigma$)
                if $r'_i$ False then
                    $R' \leftarrow R' \cup \{r'_i\}$
                end
            else
                $R' \leftarrow R' \cup \{r_i\}$
            end
        end
        $R' \leftarrow$ Merge($R'$)
        return $\bigvee_{r'_i \in R'} r'_i$
    end
else
    return $T_\rho$
end
```

Having the counterexample m_a, the Split function replaces the existing region with two new regions:

$$r_1 := (l \leq a) \wedge (a \leq m_a - 1)$$
$$r_2 := (m_a + 1 \leq a) \wedge (a \leq u)$$

Even if T_ρ already consists of multiple regions, only one region will be affected by the removal of the counterexample. In general there will be $3^n - 1$ additional regions introduced (where n is the number of parameters), while some of them might be merged later with surrounding regions.

Recursive refinement. The algorithm further checks for specification violations (see line 16 to 26) by recursively calling the refinement function on the regions (see line 19). Each recursive call is guarded by a check whether the current region r_i is compatible with the path constraint ϕ and the current patch formula (see line 17 and 18). Otherwise we cannot reason about the region. After iterating over all regions, the algorithm attempts to merge contiguous regions (see line 27), and finally, returns the disjunction of the refined parameter regions (see line 28).

5.5 Evaluation

The goal of our work is to efficiently navigate the patch space and find the correct patch that works beyond the provided test suite. We compare our technique with the related counterexample-guided inductive synthesis (CEGIS) [134, 135] because it also can be employed to navigate the patch space via patch refinement in order to generate the correct patch. Note that the above proposed technique of concolic program repair is *not* tailored to a specific class of errors. However, the low dependence on existing test cases fits well the context of repairing security vulnerabilities. Therefore, we present an empirical comparison with the state-of-the-art program repair tools Angelix [92], and Prophet [79], and also the recently proposed tool ExtractFix [40] for repairing security vulnerabilities. To highlight CPR's general repair capabilities, we also include additional subjects from the ManyBugs [46] benchmark. Furthermore, we show CPR's ability to fix logical errors for subjects from the SV-COMP benchmark [152]. All experimental data, as well as the open-source CPR tool, are available from: `https://cpr-tool.github.io/`

5.5.1 Experimental Setup

Benchmark Suite. ExtractFix [40] is a state-of-the-art vulnerability repair tool, which generates fixes for security vulnerabilities by computing a crash-free constraint using a sanitizer. The crash-free constraint is used as the oracle for patch generation, and in our case, it can serve as the program specification. We follow a different workflow by first synthesizing patches at a given fault location and then gradually improving them based on a concolic exploration. We use their benchmark, which includes real-world applications with reported security vulnerabilities, and hence, it can be used to evaluate the efficacy of our technique in repairing security vulnerabilities. The collected subjects from the ManyBugs [46] benchmark show a partial subset of programs that can be handled with our underlying concolic engine KLEE [16]. Most of these subjects represent general errors. SV-COMP [152] is a common benchmark for evaluating the effectiveness and efficiency of state-of-the-art verification techniques. We identified C programs from SV-COMP, which include reachable assertion errors and for which there is another program in the benchmark, which represents a repaired version (i.e., the assertion is present but the error is not reachable), while the repair is not just a modification of the assertion's condition, but a logical change in the program before the assertion is reached. For our experiments, we have chosen 10 programs that satisfy the stated conditions.

Experimental Setup. Our implementation of the concolic engine is an extension of KLEE [16]. All experiments are conducted on a Dell PowerEdge R530 with Intel(R) Xeon(R) CPU E5-2660 processor and 64GB RAM. We use Docker containers to exploit and repair the vulnerable applications. The experiments have been executed with the timeout of 1 hour to match the experiments of ExtractFix [40], allowing comparison with other repair tools. The language components for the synthesis are selected as needed for the specific subject and the parameters for the abstract patches have been limited to be within the range [-10,10]. For each experiment, (at least) one failing test case is provided as the initial test case. For subjects in the ExtractFix benchmark the failing test case is the exploit. For subjects in the ManyBugs benchmark there are multiple failing and passing test cases, while we provide CPR only the failing test cases. For subjects in SV-COMP we manually generate a failing test to trigger assertion errors. For ExtractFix and ManyBugs, we derive simple specifications from the programs themselves, e.g., that a program should not return an erroneous status code. The specification for the SV-COMP subjects is directly extracted based on the included assertions. For our experiments, the fault locations have been provided manually to CPR.

Our CEGIS Implementation. CEGIS comes in various forms in existing works [134, 135, 4]. We implement our own custom version of CEGIS with regard to the concepts in [135] by reusing as much components as possible from our tool CPR so that we can enable a fair comparison between the concepts with minimized impact of implementation differences. More specifically, our CEGIS implementation reuses CPR's concolic engine to provide a common path exploration for both techniques and reuses CPR's synthesizer to explore the same patch space. This custom CEGIS implementation supports the patch generation using a counterexample-guided refinement of the synthesis constraint. It starts with a concolic exploration of the input space to construct a set of path constraints. Afterwards, we synthesize a patch for the derived constraints (i.e., user-provided specification and witnessed program paths in previous concolic exploration). We then verify if the synthesized patch can produce a counterexample such that the specification is violated. If a counterexample can be found, the current patch will be thrown away, and the counterexample model is added to the synthesis constraint. The synthesizer will generate a new patch and the iteration continues until there is no further counterexample, or the patch space is covered.

It is necessary to limit the concolic exploration of CEGIS to make the techniques comparable. In our experiments, we split the overall timeout of 1 hour for CEGIS into 30 minutes initial path exploration and 30 minutes patch refinement. The conceptual difference between CEGIS and CPR is that CEGIS explores the patch space and input space *one patch / one input* at a time, while CPR explores *partitions* in both the patch space and the input space.

5.5.2 Experimental Results

5.5.2.1 Our CEGIS Implementation

Table 5.1 shows the results of the comparison between the two techniques. Column *Components* indicate the number of language components passed to our synthesizer. The sub columns *General* and *Custom* represent the number of components from the *general* synthesis language and number of *custom* components created specifically for the respective test subject. Columns $|P_{Init}|$ and $|P_{Final}|$ show the number of patches in the plausible patch space at the start of the refinement and at the end respectively. CEGIS does not maintain a patch pool like CPR, but only generates one patch that satisfies the collected constraints. However, the current patch pool size can be calculated by instructing the synthesizer to produce all currently feasible patches. $|P_{Init}|$ is for CEGIS the same as for CPR because we share the same inputs and synthesizer. Column *Ratio* shows the percentage of the patch space reduction. Column ϕ_E indicates the number of program paths *explored* for the refinement. Column ϕ_S indicates the number of program paths *skipped* during the refinement due to patch in-feasibility. Column *Correct*? indicates whether CEGIS finishes with a patch that is syntactically or semantically equivalent with the developer patch and column *Rank* shows the corresponding highest rank position.

The *N/A* values for ID 23 and 24 in Table 5.1 indicate that both CEGIS and CPR have not been able to produce any results because the execution of the test driver code resulted in an unexpected memory fault for our underlying concolic execution engine. The "-" signs for CEGIS for ID 30 mean that it was not able to generate any patch within the timeout.

Input and patch space exploration. The comparison of the *Ratio* columns in Table 5.1 shows that in 14 of 30 cases CPR can produce significantly better patch space reduction than CEGIS. In the remaining 16 cases, both perform similarly. For a few subjects, CPR resulted in 0% reduction, partly because of the loop unrolling (and hence longer paths) in symbolic execution. While this is an area we can work on, the ϕ_S column shows that CPR is already effective in combating path explosion by skipping additional paths over and above normal concolic execution. For all subjects, for which CPR produces some patch space reduction >

Table 5.1: Comparison of CEGIS vs CPR

ID	Buggy Program		Components		Our CEGIS Implementation					CPR					
	Project	Bug ID	General	Custom	$\|P_{Init}\|$	$\|P_{Final}\|$	Ratio	ϕ_E	Correct?	$\|P_{Init}\|$	$\|P_{Final}\|$	Ratio	ϕ_E	ϕ_S	Rank
1	Libtiff	CVE-2016-5321	2	3	174	174	0 %	17	✗	174	104	40%	67	77	2
2	Libtiff	CVE-2014-8128	4	3	260	260	0%	0	✗	260	260	0%	0	0	1
3	Libtiff	CVE-2016-3186	4	3	130	130	0%	13	✗	130	130	0%	13	1	11
4	Libtiff	CVE-2016-5314	4	4	199	198	1%	10	✗	199	197	1%	21	4	2
5	Libtiff	CVE-2016-9273	4	3	260	260	0%	5	✗	260	141	46%	10	2	8
6	Libtiff	bugzilla 2633	4	3	130	130	0%	66	✗	130	130	0%	109	21	8
7	Libtiff	CVE-2016-10094	4	3	130	130	0%	23	✗	130	77	41%	34	114	6
8	Libtiff	CVE-2017-7601	4	2	94	94	0%	27	✗	94	94	0%	78	107	2
9	Libtiff	CVE-2016-3623	4	3	130	130	0%	60	✗	130	100	23%	102	21	1
10	Libtiff	CVE-2017-7595	4	3	130	130	0%	10	✗	130	130	0%	18	31	1
11	Libtiff	bugzilla 2611	4	3	130	130	0%	61	✗	130	112	14%	87	15	1
12	Binutils	CVE-2018-10372	5	3	74	74	0%	9	✗	74	39	47%	25	1	33
13	Binutils	CVE-2017-15025	4	3	130	130	0%	0	✗	130	130	0%	0	0	6
14	Libxml2	CVE-2016-1834	4	3	260	260	0%	6	✗	260	260	0%	22	0	12
15	Libxml2	CVE-2016-1838	4	4	199	199	0%	4	✗	199	199	0%	4	0	10
16	Libxml2	CVE-2016-1839	5	3	65	65	0%	0	✗	65	65	0%	0	0	14
17	Libxml2	CVE-2012-5134	4	3	260	260	0%	44	✗	260	134	48%	80	271	7
18	Libxml2	CVE-2017-5969	4	3	260	260	0%	0	✗	260	154	41%	21	2	1
19	Libjpeg	CVE-2018-14498	4	3	260	260	0%	42	✗	260	128	51%	78	108	2
20	Libjpeg	CVE-2018-19664	4	3	130	130	0%	43	✗	130	130	0%	84	26	1
21	Libjpeg	CVE-2017-15232	5	3	955	955	0%	0	✗	955	955	0%	0	0	26
22	Libjpeg	CVE-2012-2806	4	3	260	259	0%	68	✗	260	145	44%	110	3	3
23	FFmpeg	CVE-2017-9992	6	3	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
24	FFmpeg	Bugzilla-1404	4	2	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
25	Jasper	CVE-2016-8691	4	3	260	260	0%	72	✗	260	96	63%	69	7	1
26	Jasper	CVE-2016-9387	5	3	65	65	0%	54	✗	65	17	74%	111	1	✗
27	Coreutils	Bugzilla 26545	5	3	1025	1025	0%	74	✗	1025	949	7%	119	2	25
28	Coreutils	GNUBug 25003	4	4	199	198	1%	114	✗	199	172	14%	196	0	6
29	Coreutils	GNUBug 25023	4	2	64	64	0%	32	✗	64	64	0%	1	2	7
30	Coreutils	Bugzilla 19784	4	3	-	-	-	-	-	770	770	0%	6	0	38

Comparison between our CEGIS implementation and CPR with regard to patch pool reduction ratio and input space reduction ratio. Benchmark: ExtractFix. The experiments have been executed with a timeout of 1 hour.

1%, it outperforms CEGIS. Furthermore, the ϕ_E columns show that CPR is also more efficient in exploring the input space: in 20 of 30 cases CPR explores more path constraints than CEGIS, in 2 cases CEGIS shows better results, and for the remaining 8 cases both perform similarly. Additionally, CPR can effectively skip infeasible path constraints (see Column ϕ_S).

Furthermore, CEGIS requires initial path exploration to construct the constraint for later patch verification. Therefore, in order to verify a patch, CEGIS uses a set of symbolic paths that capture a portion of the program specification. In contrast, our technique CPR is an anytime algorithm that uses a single program path at a time for patch refinement. Processing a single path at a time, compared to a set of paths is more efficient during constraint solving.

Identifying the correct patch. In none of our 30 test subjects CEGIS can identify a patch, which is syntactically or semantically equivalent with the developer patch (see Column *Correct*?). The reason is that as soon as CEGIS identifies a

Table 5.2: Comparison of CPR with repair tools

Benchmark	Program	#Vul	Generated Patches			Correct Patches		
			Prophet	Angelix	ExtractFix	Prophet	Angelix	ExtractFix
ExtractFix	Libtiff	11	7	7	9	1	0	6
	Binutils	2	-	-	2	-	-	1
	Libxml2	5	3	0	4	0	0	2
	Libjpeg	4	3	-	3	1	-	2
	FFmpeg	2	-	-	2	-	-	2
	Jasper	2	2	2	2	0	0	1
	Coreutils	4	2	-	2	0	-	2
Total		30	17	9	24	2	0	16

The experiments have been executed with a timeout of 1 hour [40]. For Prophet and Angelix the results show only the top-ranked patch, while for ExtractFix the results capture the only patch generated.

patch, which does not violate the specification for the previously collected path constraints, it terminates and returns this current patch. In our experiments, such a patch often is a tautology or contradiction, which can be semantically equivalent to code deletion, as the patch would enforce early termination of the program to avoid the bug location. CPR includes such patches in the patch space (as long as they do not violate any specification), but our ranking system de-prioritizes such patches (see Section 5.3.2.3). Column *Rank* shows that CPR ranks the developer patch (or a semantic equivalent) relatively high, in 20 cases in the Top-10.

5.5.2.2 Existing Program Repair Tools

CPR can be leveraged for constraint-driven repair, i.e., having just a few or no test cases, but a constraint, which can be used as a repair oracle. For this purpose, we focus on the comparison with the most recently proposed constraint-driven repair technique ExtractFix [40] and their corresponding data-set. On the data-set of ExtractFix, CPR generates the correct patch in top position for 7/30 subjects and in second position in 4/30 subjects, as shown in Table 5.1.

As already mentioned, ExtractFix uses a crash-free constraint as the guiding oracle to generate a patch. ExtractFix computes the weakest precondition for the patch by back propagating the crash-free constraint. Conceptually, ExtractFix explores the patch space using the crash-free constraint to determine the patch and then evaluates the effectiveness of the patch for the input space. In contrast, CPR can use the same crash-free constraint but explores the input space to determine the invalid values that can violate the crash-free constraint, and use this information to evaluate the effectiveness of the patch. The tool ExtractFix is also compared with conventional test-based repair tools Prophet and Angelix in [40].

Table 5.2 from [40] shows the results on the same security vulnerability benchmark. Column *#Vul* shows the count of vulnerabilities for each subject, which is in total 30. The columns *Generated Patches* and *Correct Patches* show the number of vulnerabilities, for which the techniques generated *plausible* and *correct* patches (i.e., syntactically or semantically equivalent to the developer patch).

Table 5.3: Performance of CPR on ManyBugs benchmark

ID	**Buggy Program**		**Components**		**CPR**					
	Project	Subject ID	General	Custom	$\lvert P_{Init}\rvert$	$\lvert P_{Final}\rvert$	Ratio	ϕ_E	ϕ_S	Rank
1	Libtiff	ee65c74	4	3	6	6	0%	29	90	1
2	Libtiff	865f7b2	4	3	130	130	0%	24	68	5
3	Libtiff	7d6e298	5	4	4	2	50%	7	7	1
4	gzip	884ef6d16c	5	4	4821	4821	0%	11	0	36
5	gzip	f17cbd13a1	5	4	2	2	0%	0	1	1

Performance of CPR with regard to patch pool reduction ratio and input space reduction ratio for additional subjects from the ManyBugs benchmark. The experiments have been executed with a timeout of 1 hour.

Overall, we note that ExtractFix is a customized tool for repairing security vulnerabilities which hooks into specific sanitizers, whereas ours is a general-purpose program repair machinery. Table 5.3 shows the results from test-based repair of Manybugs subjects [46] that require a general-purpose repair technique; these cannot be handled by ExtractFix. CPR can generate correct patches for all of them, by leveraging the failing tests to drive concolic path exploration. In future, it is also possible to experimentally evaluate the usage of passing tests to drive concolic exploration in CPR.

Since Prophet and Angelix are test-driven general repair techniques, in addition to the failing test case, available developer test-suite are provided to both Angelix and Prophet (the programs in Table 5.2 come with test-suites from developers). ExtractFix and CPR do not need additional tests.

Angelix *and* Prophet In contrast to our approach, ExtractFix is driven only by the initial test case while Angelix and Prophet both use additional developer test cases. Despite being provided additional test cases, both Angelix and Prophet cannot produce many *correct* patches. Prophet can only identify correct patches for 2 of the vulnerabilities and ANGELIX is not able to correctly fix any of them, *as the top-ranked patch.* Most of the correct patches represent updated or inserted conditions, which are in the search space of both techniques. However, as mentioned in EXTRACTFIX [40], the developer-provided tests for this benchmark are very limited, which may lead to overfitting patches. Therefore, ANGELIX cannot generate a rich specification for synthesis, and PROPHET suffers from a large search space. PROPHET and ANGELIX have the potential to repair more vulnerabilities if more tests are available, and if more of their ranking is examined, i.e., beyond the top-ranked patch.

5.5.2.3 Fixing Logical Errors

We further evaluate CPR on its capability to repair logical errors of a program provided as assertions or rich-text comments on the source code.

Therefore, we investigate the possibility of repairing programs beyond simple oracles such as crash-freedom. We evaluate the efficacy of CPR in fixing logical errors on subjects from the SV-COMP benchmark, which is popular for automated program verification and provides such program specifications. As mentioned earlier, for our chosen SV-COMP programs the developer provided patch is available in the form of another program (so we can check whether CPR produced the correct patch), and the developer provided patch is not merely a change of the assertion but involves a change in the functionality.

Table 5.4: Performance of CPR for logical errors in SV-COMP

ID	**Subject**	**Components**		**CPR**					
		General	Custom	$\|P_{Init}\|$	$\|P_{Final}\|$	Ratio	ϕ_E	ϕ_S	Rank
1	loops/insertion_sort	4	3	260	132	49%	120	0	1
2	loops/linear_search	4	3	260	127	51%	109	17	1
3	loops/string	2	3	676	676	0%	37	0	2
4	loops/eureka	5	3	29	29	0%	107	27	3
5	loops-crafted-1/nested_delay	4	3	260	117	55%	9	8	4
6	loops/sum	4	3	260	236	9%	116	0	1
7	array-examples/bubble_sort	4	3	260	144	45%	34	19	2
8	array-examples/unique_list	1	2	5	4	20%	134	11	1
9	array-examples/standard_run	4	3	260	126	52%	68	41	1
10	recursive/addition	5	3	38	14	63%	138	1	4

Performance of CPR with regard to patch pool reduction ratio and input space reduction ratio for the repair of logical errors in SV-COMP. The experiments have been executed with a timeout of 1 hour.

Table 5.4 presents the results. The meaning of the columns is similar to Table

5.1 in Section 5.5.2.1. For all subjects, CPR can identify correct patches in the patch pool. Furthermore, due to the efficient space exploration, CPR achieves a patch space reduction ratio of up to 63 %. Only for one subject (*loops/eureka*) CPR was not able to produce any patch space reduction. The reason is that the assertion in the program was not strong enough to identify violations. However, CPR still has been able to rank the correct patch on position 3. In fact, for all of the 10 subjects CPR can rank the correct patches in the Top-10 and for five of them as Top-1.

5.5.2.4 Internal Evaluation of CPR Components

Parameter Range As mentioned in our *Experimental Setup* section, the parameters for the abstract patches in our experiments are limited within the range [-10, 10]. We conducted additional experiments to show the effects of other ranges.

Table 5.5: Impact of different parameter ranges on the repair success of CPR

Buggy Program		**Parameter**	**CPR**					
Project	Bug ID	**Range**	*#Iter.*	ϕ_E	$\|P_{Init}\|$	$\|P_{Final}\|$	Ratio	Rank
Jasper	CVE-2016-8691	[-1, 1]	70	68	44	15	66%	1
		[-10, 10]	70	69	260	96	63%	1
		[-100, 100]	70	79	2420	907	63%	1
Libtiff	CVE-2016-10094	[-1, 1]	35	34	22	10	55%	-
		[-10, 10]	35	34	130	77	41%	6
		[-100, 100]	27	26	1210	887	27%	6

Benchmark: selection of ExtractFix. The experiments have been executed with a timeout of 1 hour.

The results in Table 5.5 show that the number of initial patch candidates ($|P_{Init}|$) is growing with a larger parameter range. The effort for the initial patch pool construction is not largely affected because the concrete values for the parameters are not enumerated but abstracted in the range. The ranking of the correct patch itself is not necessarily affected as our experiments show. For *Jasper/CVE-2016-8691* the correct patch is correctly identified after the first iteration. For *Libtiff/CVE-2016-10094* the parameter range needs to include the constant 4 so that CPR can identify the correct patch. With a too narrow range like $[-1, 1]$ CPR cannot identify the correct patch.

Table 5.6: Path exploration of CPR

Benchmark	**Avg. PatchLoc Hit**	**Avg. BugLoc Hit**
ExtractFix	74.36%	40.23%
ManyBugs	57.14%	65.15%
SV-COMP	76.33%	79.08%

Average ratio of the number of generated inputs that hit the patch and bug location.

Input Generation The additional generation of inputs is an essential part of our path exploration phase (see Section 5.3.2.2). Our search heuristics drive the input generation to the bug location. Hitting the bug location is crucial, not only to rule out patches, but also to improve the patch ranking. Table 5.6 shows how often our generated inputs hit the patch and bug location on average. The results show that to a large extent our generated inputs do exercise the patch and bug location. However, for the ExtractFix benchmark hit count for the bug location is comparably low with 40.23%. In contrast to the SV-COMP subjects, where the inputs represent primitive data types, the ExtractFix subjects require complex input structures like images or XML files. Our input generation does not use an application-specific input grammar, which could lead to a significant improvement.

Patch Ranking The changes in our ranking are based on whether the generated inputs exercise the patch and bug location under the specific patches. For many subjects the ranking of the correct patch is already very high after the first few iterations, and is not changed later. Our path exploration starts with inputs that exercise paths that are close to the path of the failing test case: hitting the bug location is more likely for those inputs. In some subjects, the ranking improved gradually over the repair time, e.g. *Coreutils/Bugzilla 26545* starts with the correct patch ranked at position 104 and it improves to 25 (after 65^{th} iteration). Change in ranking can happen due to patch candidates violating specification in the new paths.

5.5.3 Threats to Validity

In the formulation of our repair algorithm, as well as in our experiments, we assume that the *correct* patch is included in the initial patch pool P. This is only the case, if our synthesis language/grammar covers this patch. In general, this assumption might not hold. In such a case, our ranking allows us to still present the most *promising* patches, which can only repair the program for a portion of the input space. Our approach currently focuses on repairing boolean and integer expressions. In future we want to extend our work to repair complete assignments as well as side-effect free function calls.

Our approach requires some ingredients that differs from existing program repair strategies: the user-provided (partial) specification and the fault locations (see the input description in Section 5.3.2). The specification allows us to reason about many program inputs going beyond a test suite. Other techniques rely on bug templates, sanitizers, existing test cases, or probabilistic models to reason about the correct behavior. Our specifications are lightweight, and our experiments show that even simple specifications can be used to rule out overfitting patches in an incremental manner. The fault location information is an input to our approach, which can be derived from statistical fault localization. Test-based repair tools may use a set of fault locations, while our approach currently works with one fault location at a time.

5.6 Summary

In this chapter, we take a look at the problem of repairing security vulnerabilities using a user-provided specification. We note that the patches produced by current program repair techniques may not even ensure very basic notions of correctness such as crash-freedom, or assertions, even when such simple specifications are readily available. Our concolic exploration incorporates a user-provided program-specification to identify overfitting patches that are plausible but do not satisfy the specification for at least one of the generated inputs. Furthermore, by removing incorrect but plausible patches we shrink the patch space and increase the ranking of the correct patch, alleviating patch overfitting which is a necessity to find the correct repair for security vulnerabilities.

Chapter 6

Vulnerability Repair via Patch Backporting

Fixing security vulnerabilities does not always entail a passing test-case that was previously failing. In practice, some of the security vulnerabilities reported may not contain a proof of concept i.e. an exploit to reproduce the bug. In this chapter we look into the evolution history of a software, to generate plausible patches to existing un-patched security vulnerabilities. More specifically we study the problem of automated patch backporting to repair security vulnerabilities in old versions of the same software. We select the Linux kernel project as a case-study for this problem, where the impact of an automated backporting technique is high, especially if there is a security vulnerability that is exposed for exploitation. Whenever a bug or vulnerability is detected in the Linux kernel, the kernel developers will endeavour to fix it by introducing a patch into the mainline version of the Linux kernel source tree. However, many users run older "stable" versions of Linux, meaning that the patch should also be "backported" to one or more of these older kernel versions. This process is error-prone and there is usually a long delay in publishing the backported patch. In this chapter, we investigate how adapting program synthesis technique for program transformations can meet the needs of backporting patches in Linux kernel from the mainline to old stable versions. The chapter starts with an overview of the backporting problem, followed by an empirical study on the backporting effort for Linux kernel. It continues with an illustrative example to demonstrate the challenges of backporting a patch, and continues with the technical details of our proposed approach. Afterwards, the chapter concludes with a preliminary evaluation, showing the effectiveness of our proposed approach in successfully backporting patches from the mainline version of Linux into an older stable version.

6.1 Overview

Although the mainline version of the Linux kernel shares a common codebase with older versions, they typically diverge over time as different features and fixes are added to the latest branch. As a result, when a bug is patched in the mainline version, the patch is often not directly applicable to another version. Given a patch created for the latest version, backporting involves identifying the correct patch location and adapting the patch to an older version. Backporting is typically done manually by a developer, on a case-by-case basis. The manual process of backporting is error-prone and there is usually a long delay in publishing the backported patch. This becomes critical when we consider security patches.

To understand the importance and challenges of backporting patches, we first conduct an empirical study on the Linux kernel versions spanning 2011-19. We found that (1) 51,663 patches have been backported from the mainline to old versions, representing around 8% of all the commits to the mainline version, and (2) the backporting process typically took more than one month. Moreover, backporting patches is not simple copy and paste, as it may involve changing patch locations, changing the namespace (the variable or function names used in different versions), and modifying the code logic and structure. These findings indicate that automatically backporting patches is important but challenging.

Existing program transformation techniques can potentially be applied to automate the backporting process. Automated program transformation [93, 77, 54, 113, 6, 10, 120] infers transformation rules from human-written patches, and then applies the inferred rules to an unforeseen codebase. These approaches have been used to fix software bugs (e.g. GetaFix [6] and Phoenix [10]), automate repetitive edits (e.g. Refazer [113] and Lase [93]), etc. However, they have two main limitations: 1) they learn transformation rules from multiple human-written patches, which are not always available in reality; 2) the program transformation techniques for fixing software bugs, such as GetaFix [6], Phoenix [10] and Genesis [77], infer transformations from the patches of different applications, so they can only learn general transformation patterns shared by multiple applications, e.g., inserting null checks, fixing API usage errors. These limitations prevent the above techniques from effectively backporting Linux kernel patches. In a backporting setting, 1) there is usually only one available patch (the one introduced in the mainline version) and 2) most kernel patches are specific to the kernel and the fix pattern cannot be learned from other projects. Although GenPat [54] and Sydit [94] require only one example, GenPat requires a large codebase to provide statistical information on how to generalize the example, and Sydit simply generalizes all identifiers and edit positions which may lead to false positives.

The main challenge of synthesizing transformation rules from human patches lies in inferring a proper generalization. An *under-generalized* transformation rule can lead to false negatives: it cannot generate patches for some locations that should be patched. An *over-generalized* transformation rule produces false positives: it may generate patches for some locations that should not be patched. The generalization problem becomes more serious when only one human patch is available. Consider the following example patch that fixes an off-by-one error by changing < to <=:

$$\texttt{if}\ (chunk\_end + *\,ch \texttt{ < } skb) \mapsto \texttt{if}\ (chunk\_end + *ch \texttt{ <= } skb)$$

In general, it is hard to infer whether to 1) generalize the variable, e.g. ch, 2) generalize the dereference operation $*ch$, or 3) generalize the whole left operand of the comparison.

Different from existing program transformation systems [6, 10, 77] that transform patches across different projects, our goal is to transform patches between different versions of the same project. Different versions of the same project share similar expressions, algorithms, namespaces, etc. Our main insight is that the similarities between versions can guide us in synthesizing properly generalized transformation rules. Suppose $\texttt{v}_{\texttt{mainline}}$ is the original mainline version targeted by developer patch and $\texttt{v}_{\texttt{old}}$ is the old version to which the patch should be backported. For the above example, we might observe that $\texttt{v}_{\texttt{mainline}}$ and $\texttt{v}_{\texttt{old}}$ use many variables (e.g. *chunk_end* and *skb*), expressions and algorithms identically. In the function affected by the patch, we also might observe the following: matched statements:

$$\texttt{v}_{\texttt{mainline}} : \texttt{skb\_pull}(skb, *ch)$$

$$\texttt{v}_{\texttt{old}} : \texttt{skb\_pull}(skb, sctp\_chunkhdr\_t)$$

These matching statements suggest that $*ch$ should correspond to *sctp_chunkhdr_t* in the old version. To backport this patch from the mainline to the old version, when synthesizing the transformation rule, this observation can guide us to generalize $*ch$ while keeping the other elements concrete.

In this chapter, we adopt the program synthesis technique for program transformations and investigate how it can be adapted to meet the needs of backporting patches from the mainline to old stable versions. Specifically, we synthesize a transformation rule from the $\texttt{v}_{\texttt{mainline}}$ patch and apply the transformation rule to multiple old versions ($\texttt{v}_1, \texttt{v}_2, ...\texttt{v}_\texttt{n}$). First, based on the single $\texttt{v}_{\texttt{mainline}}$ patch, we represent it as a transformation rule R using a *Domain Specific Language.* Since transformation rule R is specific to the given patch, we propose a notion of *partial program transformation rule* R^p, which allows certain fragments of R^p to be generalized according to the context in which R^p is applied. For the above example, the partial transformation rule could be:

$$\texttt{if}(\mathit{chunk\_end}{\sim}\texttt{true} + *\mathit{ch}{\sim}\texttt{true} \texttt{ < } \mathit{skb}{\sim}\texttt{true}) \mapsto$$
$$\texttt{if}(\mathit{chunk\_end} + *\mathit{ch} \texttt{ <= } \mathit{skb})$$

where identifiers (e.g. *chunk_end*) and an expression $*ch$ are marked as flexible ($\texttt{e}{\sim}\texttt{true}$ means $\texttt{e}$ is flexible). Second, we determine how to generalize these flexible elements according to the alignment of $\texttt{v}_{\texttt{mainline}}$ and $\texttt{v}_\texttt{i}$, for each targeted older version $\texttt{v}_\texttt{i}$. This alignment models the matched code elements in $\texttt{v}_{\texttt{mainline}}$ and $\texttt{v}_\texttt{i}$ with respect to the file, function, expression, namespace, etc. The main insight is that similarities and differences of $\texttt{v}_{\texttt{mainline}}$ and $\texttt{v}_\texttt{i}$ modeled by the alignment can guide us to decide which elements should be generalized. This enables us to find an appropriate generalization according to the target version in an on-demand manner.

6.2 Empirical Study

We conduct an empirical study of changes in the Linux kernel to better understand the extent and characteristics of patch backporting. Specifically, our study answers the following research questions:

RQ1: How many patches are backported per release? What percentage of patches are backported?

RQ2: How long does it take to backport patches?

RQ3: How do developers backport patches? Can the patches be applied directly, or do developers need to modify the patches, and if so how do they modify the patches?

In our study, we investigated 46 versions (v3.1 to v5.5) of the Linux kernel covering nine years (2011-2019). In total, we collected 633,860 commits submitted to the mainline version of the kernel and 144,437 commits that backport a mainline patch to an older stable version. We focus on changes made to source code files and exclude commits that change other types of files (e.g., configuration files).

6.2.1 Percentage of Backported Patches

We analyze the percentage of backported patches (commits) out of all released patches. For each release version $\texttt{v}_{\texttt{release}}$, we compute the number of patches introduced in $\texttt{v}_{\texttt{release}}$ and cross-reference with patches that were backported to older versions. Figure 6.1 shows the distribution of the percentage of patches that were backported for each $\texttt{v}_{\texttt{release}}$. The distribution ranges from 3.87% - 16.29%, and on average, 8% of patches for a release have been backported to at least one older version. In total, among 633,860 mainline patches in released versions, 51,663 have been backported to older versions. A patch is only backported if it fixes an important bug or is required to enable fixing an important bug [47].[1] As users rely heavily on the old stable versions (much more than the mainline), backporting all those patches is critical.

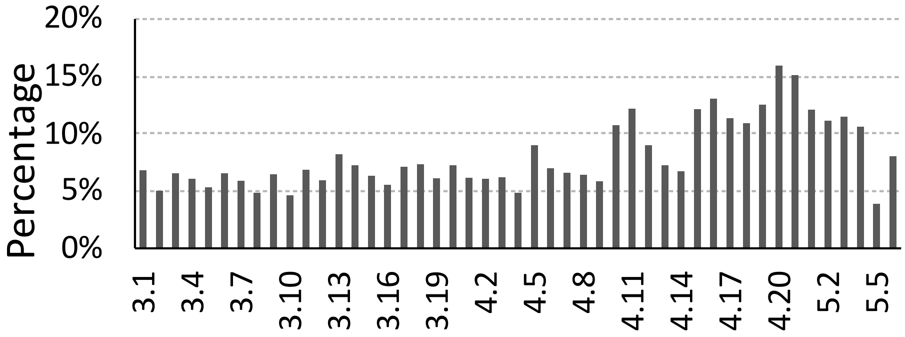


Figure 6.1: The distribution of backported patches per release

[1] S small percentage of patches add new device properties. These are considered to introduce very low risk, due to the simplicity of the change, and high value.

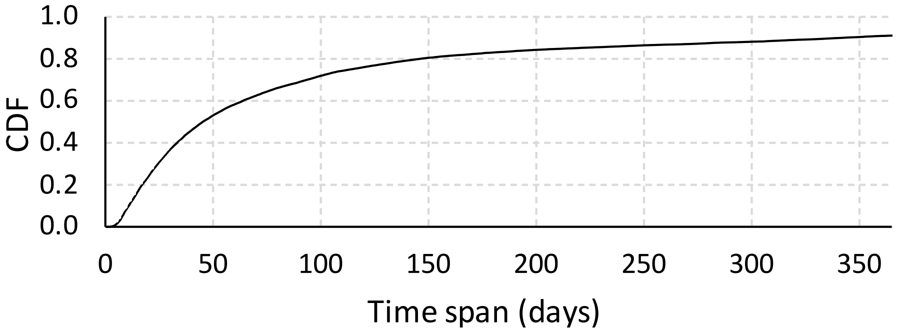


Figure 6.2: Cumulative distribution of patch backporting time

51,663 patches, accounting for 8% of all patches (commits), were backported to older versions during 2011-2019.

6.2.2 How long does it take to backport a patch?

We investigate the delay between the time when a patch is committed to the mainline and when it appears in all relevant stable versions. We measure this delay by computing the difference between the commit date of the patch in the mainline and the commit date of the last backported patch. For example, the patch with commit ID db4175ae[2] was committed on 15 Jul 2014, and it was backported to five stable versions (v3.2, v3.10, v3.12, v3.14 and v3.15). The last backported patch (commit ID 5248ee65) was committed to v3.2.63 on 13 Sep 2014. Hence, the time to backport this patch is 15 Jul 2014 – 13 Sep 2014, which is 60 days.

Figure 6.2 shows a cumulative distribution function (CDF) for the time of backporting patches in days. For simplicity, we only show the data for a duration of up to a year. 80% of patches took more than 20 days, while around 50% took more than 46 days. We also found that around 10% of backported patches took more than 365 days, amounting to 4844 commits. As some bugs may be security critical, the longer time it takes to backport patches, the higher possibility that such bugs can be exploited by malicious attackers. These results indicate the necessity to accelerate the patch backporting process and motivate us to design approaches to automate it.

[2] The details of each commit can be found in https://kernel.googlesource.com/pub/scm/linux/kernel/git/stable/linux-stable /+/COMMIT_ID^!

Around 50% of backported patches took more than 46 days to be backported from the mainline to old stable versions.

6.2.3 How does a developer backport patches?

To investigate the patch backporting effort required for a developer, we manually inspect the backported patches. We choose to study only the patches backported from a specific version (i.e., v3.8). We label commits based on the difficulty of backporting:

- *Type-I (no changes)*: the backported commit does not require any change from the original patch;
- *Type-II (only patch location changes)*: the patch location(s) (e.g. the containing filename(s) and function name(s), line number(s)) are different in the mainline and the old versions;
- *Type-III (only namespace changes)*: the original patch is adapted by modifying variable names, function names, etc.;
- *Type-IV (patch location & namespace changes)*: the original patch is adapted by changing both the patch location *and* namespace;
- *Type-V (logical or structural change)*: other changes are needed, such as adding extra code or removing irrelevant code.

Table 6.1: Developer effort in backporting patches for the Linux kernel project

Label	Description	Count	Percentage
Type-I	no changes	149	22.9%
Type-II	only patch location changes	431	66.3%
Type-III	only namespace changes	0	0%
Type-IV	location & namespace changes	20	3.1%
Type-V	logical and structural changes	50	7.7%

Table 6.1 shows the results of our manual analysis. In our analysis, most of the backported patches are Type II, which require changes to the patch locations. More than 10% of them are Type-IV or Type-V, which represent the more challenging cases. If a patch has been backported to multiple versions, we notice that

```
int unix_read(struct
   unix_state *state) {   ......
+  scm_destroy(...);
}
```

```
int unix_revmsg(struct
   unix_state *state) {   ......
+  scm_destory(...);
}
```

(a) Patch location changes when backporting from v4.5 to v3.2

```
- create_seq("typeinfo",
    0444, NULL,
    &pageinfo_op);
+ create_seq("typeinfo",
    0400, NULL,
    &pageinfo_op);
```

```
- create("typeinfo",
    S_IRUGO, NULL,
    &pageinfo_fops);
+ create("typeinfo",
    0400, NULL,
    &pageinfo_op);
```

(b) Namespace changes when backporting from v5.5 to v3.16

```
if (dev->vendor==ID_INT) {
   ...
+  xhci->quirks |= XHCI_AVOID_BEI;
}
```

```
+ if (dev->vendor==ID_INT)  {

+   xhci->quirks |= XHCI_AVOID_BEI;
+ }
```

(c) Structure changes when backporting from v4.0 to v3.2

Figure 6.3: Different types of changes in backporting a patch

the patches backported to the oldest version are more likely to be Type-IV or V, indicating the challenges of backporting patches to very old versions. According to our manual inspection, the patch location and namespace changes (Type-II, III, IV) are easier to automate, while automating the Type-V changes is more challenging.

Figure 6.3 shows three simplified backported patch snippets. Figure 6.3a shows a backported patch that changes the patch location. This patch was first introduced in the function `unix_read` in v4.5 to fix a memory leak bug, and it was then backported to v3.2, but to a different function `unix_revmsg`. To backport this patch, the developer must manually find the correct function in the target version.

Figure 6.3b shows a backported patch that requires namespace changes. This patch was introduced in v5.5 to fix a vulnerability, and backported to v3.16, by changing the API call from `create_seq` to `create`, and the arguments from `0444` and `pageinfo_op` to `S_IRUGO` and `pageinfo_fops`. Finally, Figure 6.3c shows a patch that requires structural changes. This patch added a quirk `XHCI_AVOID_BEI` to v4.0 under if-condition `if (dev->vendor == ID_INTEL)`. This if-condition does not exist in Linux v3.2. So, to backport this patch, the developer needs to backport this if-condition as well.

> When backporting patches, a developer needs to find correct patch locations, change the namespace, and modify the program logic and composition of the patch.

6.3 Example

Figure 6.4 depicts a simplified bug-fixing patch and its corresponding backported patches. This patch was first introduced in v5.1 (Figure 6.4a) and fixed a fault in the kernel paging request handler. The patch changes an immediate return to a goto to take advantage of the shared error handling code at the end of the function. It also stores the original return value in a variable used by this shared code. This patch was backported to eight stable versions (v3.16, v3.18, v4.4, v4.9, v4.14, v4.19, v4.20, and v5.0). The backported patches for v4.9 and v3.16 are shown in Figures 6.4b and 6.4c, respectively. We make two observations, 1) the if-condition (highlighted in Figures 6.4b and 6.4c) of the backported patches is not the same as the if-condition of the mainline version v5.1 and 2) there is no returned value in v3.16

Given the patch *p* shown in Figure 6.4a, a developer needs to take the following steps to backport *p* to older versions. First, the developer needs to analyse *p* to understand the surrounding context where *p* is applied and understand how *p* changes the program. Second, since the mainline version $\texttt{v}_{\texttt{mainline}}$ and target version $\texttt{v}_{\texttt{i}}$ are not the same with respect to the affected code, the developer needs to analyze their similarities and differences to find the correct location in $\texttt{v}_{\texttt{i}}$ at which to apply *p*. At the same time, the developer may need to adjust *p* according to the context of $\texttt{v}_{\texttt{i}}$. Last, the developer produces a patch for the target version. FixMorph tries to automate this process via transformation rule synthesis. Specifically, FixMorph takes the whole if-statement as *p*'s surrounding context and synthesizes a partial transformation rule R^p. The transformation rule is represented using a domain-specific language, that will be explained in Section 6.4.2. For simplicity, we show

```
 if (!gcells->cells || skb_cloned(skb) || netif_elide_gro(dev)){
-    return netif_rx(skb);
+    res = netif_rx(skb);
+    goto unlock;
 }
```

(a) The patch introduced in v5.1 (commit 2a5ff07a)

```
 if (!gcells->cells || skb_cloned(skb) || !(dev->features & NETIF_F_GRO) ) {
-    return netif_rx(skb);
+    res = netif_rx(skb);
+    goto unlock;
 }
```

(b) Backported patch from v5.1 to v4.9 (commit 7cbb0ab1)

```
 if ( !cell  || skb_cloned(skb) || !(dev->features & NETIF_F_GRO) ) {
    netif_rx(skb);
-   return;
+   goto unlock;
 }
```

(c) Backported patch from v5.1 to v3.16 (commit 415f08eb)

Figure 6.4: Sample backporting task

R^p for this example as follows:

$$\begin{aligned}
&\texttt{if}(t_1 \mathbin{||} t_2 \mathbin{||} t_3)\ \{\ \texttt{return}\ m_1(a_1);\ \} \mapsto \\
&\qquad\qquad \texttt{if}(t_1 \mathbin{||} t_2 \mathbin{||} t_3)\ \{\ v_1{=}m_1(a_1);\ \texttt{goto}\ l_1;\} \\
&\ \texttt{where}\ t_1.\texttt{type}{=}\texttt{bool}\ \wedge\ t_1.\texttt{code}{=}\text{``}!gcells{-}{>}cells\text{''} \\
&\qquad\qquad \wedge\ t_2.\texttt{type}{=}\texttt{bool}\ \wedge\ t_2.\texttt{code}{=}\text{``}skb\_cloned(skb)\text{''} \\
&\qquad\qquad \wedge\ t_3.\texttt{type}{=}\texttt{bool}\ \wedge\ t_3.\texttt{code}{=}\text{``}netif\_elide\_gro(dev)\text{''} \\
&\qquad\qquad \wedge\ a_1.\texttt{type}{=}\texttt{struct}{*}\ \wedge\ a_1.\texttt{code}{=}\text{``}skb\text{''}\ \wedge\ \ldots
\end{aligned}$$

The partial transformation rule R^p keeps the keywords (e.g. `if`) and some operators that affect high level transformation structures (e.g. ||) fixed and leaves the other elements in R^p as *flexible* for follow-up adjustment. In this case, the expressions t_1, t_2, t_3, m_1, v_1 and a_1 are marked as *flexible*, meaning that the constraints on them can be relaxed. In this way, R^p allows certain expressions to be generalized, so that FixMorph can determine the correct level of generalization according to the target version by relaxing different expressions.

How does FixMorph decide which flexible expressions should actually be relaxed for a given target version? To backport p from v5.1 to v4.9, the original R^p cannot be directly applied. The first two boolean expressions (corresponding to t_1 and t_2 in R^p) are the same in v5.1 and v4.9, but the third expression is different. Therefore, FixMorph relaxes the constraints on t_3 by dropping the constraint on t_3.code, allowing t_3 to be a different boolean expression. This leads to the following rule, which is used for backporting to v4.9.

$$\begin{array}{l}
\texttt{if}(t_1 \mid\mid t_2 \mid\mid t_3)\ \{\ \texttt{return}\ m_1(a_1);\ \} \mapsto \\
\qquad \texttt{if}(t_1 \mid\mid t_2 \mid\mid t_3)\ \{\ v_1{=}m_1(a_1);\ \texttt{goto}\ l_1; \} \\
\texttt{where}\ t_1.\texttt{type}{=}\texttt{bool}\ \wedge\ t_1.\texttt{code}{=}\text{“}!\mathit{gcells}{-}{>}\mathit{cells}\text{”} \\
\qquad \wedge\ t_2.\texttt{type}{=}\texttt{bool}\ \wedge\ t_2.\texttt{code}{=}\text{“}\mathit{skb\_cloned}(\mathit{skb})\text{”} \\
\qquad \wedge\ t_3.\texttt{type}{=}\texttt{bool} \\
\qquad \wedge\ a_1.\texttt{type}{=}\texttt{struct}{*}\ \wedge\ a_1.\texttt{code}{=}\text{“}\mathit{skb}\text{”}\ \wedge\ \ldots
\end{array}$$

Backporting to v4.9 required relaxing t_3. Backporting p to v3.16 (see Figure 6.4c) requires relaxing both t_1 and t_3. Further, backporting the patch to v3.16 requires a post-processing adjustment for the transformation, which will be explained in Section 6.4.5. We omit the details of the relaxed rule for v3.16.

In this example, FixMorph needs to generate different levels of generalization (by relaxing different flexible expressions) to backport the patch to the different versions. The most generalized transformation rule (generalize all flexible expressions t_1, $t_2, \ldots$) is able to transform all the versions (except for the post-processing adjustment). However, it will produce many false positives, i.e., incorrectly transforming some if-statements that should not be transformed, e.g. `if(a || b || c) return foo(i)`.

6.4 Methodology

6.4.1 Preliminaries and Problem Statement

Typed Abstract Syntax Trees. An *Abstract Syntax Tree* (AST) is a tree representation of the syntactic structure of source code. A *typed* AST associates each tree node with one or more *attributes*, including type information (e.g., `int`, `bool`, etc.), code, filename, function name, etc. We denote the set of typed ASTs as $\mathbb{T}$.

Transformation Rule A transformation rule $R : \mathbb{T} \rightarrow \mathbb{T}$ formulates how to transform a $\mathbb{T}$ to another $\mathbb{T}$. Rule R can be represented as a pair (`guard`, `transformer`) similar to [**refazer**, 95] defined as follows:

- `guard`: $\mathbb{T} \rightarrow$ `Boolean`: `guard` is a conjunction of predicates over AST nodes. Basically, a `guard` tests the type, code and other attributes of an AST node and returns a Boolean value representing whether the node satisfies its predicate or not;

- `transformer`: $\mathbb{T} \rightarrow \mathbb{T}$: `transformer` takes an input $\mathbb{T}$ and constructs another $\mathbb{T}$. It is built from two underlying operations: (1) `select`: returns an existing node from input $\mathbb{T}$ satisfying a given *guard*, and (2) `construct`: returns a new node constructed from a specific node kind, attributes, and children.

Essentially, the rule `guard` determines which AST sub-node should be transformed, and the `transformer` determines how the sub-node should be transformed. Thus, for $t \in \mathbb{T}$, we have $R(t) = \texttt{transformer}(t)$ when $\texttt{guard}(t)$ is `true`, otherwise, $R(t)$ is $\perp$.

Transformation Rule Synthesis Given an input domain $\mathbb{I}$ and an output domain $\mathbb{O}$, *program synthesis* takes a set $\{i_0 \mapsto o_0, ..., i_n \mapsto o_n\}$ of input-output pairs and synthesizes a program $P : \mathbb{I} \rightarrow \mathbb{O}$ such that $P(i_k) = o_k$ for $k \in 0...n$. For this chapter, $\mathbb{I}$=$\mathbb{O}$=$\mathbb{T}$, and thus the synthesized program can serve as a *transformation rule* that transforms an input $\mathbb{T}$ to output $\mathbb{T}$. In general, the aim is to synthesize a transformation rule (`guard`, `transformer`) that is the generalization of the concrete transformations, so that $\texttt{guard}(i_k) = \textit{true}$ and $\texttt{transformer}(i_k) = o_k$ for all $k \in 0...n$. Many existing synthesis engines, e.g., Refazer, produce the most specific generalization. That is, given a single input-output pair, those techniques do not generalize anything.

Patch Backporting Problem A patch p can be thought of as a concrete transformation from one $\mathbb{T}$ to another $\mathbb{T}$. Thus, to backport a patch from mainline version $\texttt{v}_{\texttt{mainline}}$ to old stable versions $\{\texttt{v}_1, \ldots \texttt{v}_\texttt{n}\}$, FixMorph first synthesizes a transformation rule $R : \mathbb{T} \to \mathbb{T}$ using the $\texttt{v}_{\texttt{mainline}}$ patch p, and then applies R to $\{\texttt{v}_1, \ldots \texttt{v}_\texttt{n}\}$ to produce patches. Since R simply expresses the given concrete transformation ($\texttt{v}_{\texttt{mainline}}$ patch p), we find that R is *overfitting*. That is, R can be applied to $\texttt{v}_{\texttt{mainline}}$, but often cannot be directly applied to the older versions $\{\texttt{v}_1, \ldots \texttt{v}_\texttt{n}\}$.

Partial Transformation Rule To address the overfitting problem, we introduce a notion of *partial transformation rule* R^p. The rule R^p annotates certain predicates as *flexible*. Intuitively, a partial transformation rule R^p is a flexible generalization of the given concrete transformation (i.e. the patch). This flexibility allows R^p to be generalized in an "on-demand" manner according to the context in which R^p is applied. Hence, FixMorph finds an appropriate level of generalization for each old version $\texttt{v}_\texttt{i}$, allowing backporting to $\texttt{v}_\texttt{i}$.

6.4.2 A DSL for Backporting Patches

Refazer performs transformation rule synthesis by searching over a Domain-Specific Language (DSL) for specifying transformation rules. FixMorph extends this DSL to the language $\mathcal{L}_T$ shown in Figure 6.5 (differences are highlighted in grey) to address the needs of patch backporting. The differences are as follows.

First, to allow on-demand generalization, we allow predicates to be marked with a `flexible` annotation, denoted by `pred` $\sim$ `flexible`, where `flexible` is a Boolean value. If a predicate can be relaxed, its corresponding `flexible` annotation will be `true`, otherwise it will be `false`. Second, existing synthesis frameworks focus on the local context (e.g., node kind). However, when backporting patches between different versions, we find that the *global* context (e.g., the file name and function name) can also help guide the backporting process. In general, a patch will most likely be backported to a file and function with the same name as in $\texttt{v}_{\texttt{mainline}}$. To support this feature, we add two predicates `InFile` and `InFunction` to our DSL $\mathcal{L}_T$. Finally, since FixMorph is built on top of typed ASTs, we also add a type checking (`HasType`) predicate to $\mathcal{L}_T$.

```
rule         := (guard, transformer)
guard        := pred ∼ flexible | Conjunction(pred, guard)
pred         := IsKind(node, kind)
                | Attribute(node, attr) = value
                | Not(pred)
                | HasType(node, type)
                | InFile(node, fileName)
                | InFunction(node, functionName)
flexible     := true | false
transformer  := select | construct
construct    := Tree(kind, attrs, childrenlist)
childrenlist := EmptyChildren | select | construct
                | Cons(construct, childrenlist)
                | Cons(select, childrenlist)
select       := Match(guard, node)
node         := ...
```

Figure 6.5: Domain-specific language in FixMorph for transformation rules

6.4.3 Transformation Rule Synthesis

In this section, we describe how to synthesize a partial transformation rule R^p from a given $\texttt{v}_{\texttt{mainline}}$ patch p. Given patch p, FixMorph first builds two ASTs $t_i, t_o \in \mathbb{T}$ representing the code before/after the application of p. Essentially, p is represented as an AST transformation $t_i \overset{p}{\mapsto} t_o$. A typical patch p will only affect some subsets of the complete code, e.g., some specific lines, statements, or functions. Rather than representing p as a *global* transformation over the entire file (or files), we restrict t_i and t_o to the *local* AST nodes changed by p as well as some surrounding context. Our approach is analogous to the *context diff formats* supported by the standard `diff` and `patch` tools, where the patch p includes not only the changed lines, but also some surrounding unchanged lines for context. The context serves as a reference point and allows for the patch to be applied even if other unrelated parts of the code have been modified. Since we aim to backport patches to older versions with other modifications applied, our motivation is similar. For the context, FixMorph takes the parent and all siblings of any AST node changed by p. For example, the patch shown in Figure 6.4a changes a branch of an if-statement. FixMorph, therefore, takes its parent node, i.e., the if-statement, as the surrounding context. Thus, the patch p is represented as an AST transformation over the if-statement rather than specific changed nodes. In addition to the AST context, FixMorph also includes other forms of context in the guard, such as the file and function name of the patch location.

Algorithm Given an input-output pair (t_i, t_o) extracted from patch p, FixMorph first translates p to a transformation rule in the form $(R_{\texttt{guard}}, R_{\texttt{transformer}})$, which is specified using the $\mathcal{L}_T$ DSL in Figure 6.5. In particular, the synthesis engine first synthesizes the most specific $R_{\texttt{guard}}$ that satisfies $R_{\texttt{guard}}(t_i) = \mathit{true}$. This is essentially a conjunction of all $\mathcal{L}_T$ predicates satisfied by t_i. Similarly, the synthesis engine synthesizes a $R_{\texttt{transformer}}$ that implements the transformation $t_i \mapsto t_o$. However, the produced transformation rule is overfitting to the given input-output pair (t_i, t_o), and does not generalize to others. Therefore, instead of directly using $R_{\texttt{guard}}$ and $R_{\texttt{transformer}}$, FixMorph produces a partial transformation by marking one or more predicates used by them as being flexible. Specifically, FixMorph marks the predicates of $\mathtt{R}_{\mathtt{guard}}$ as flexible to allow relaxing the requirements in finding locations to apply the patch. FixMorph marks the predicates of `guard` used by `select` operators as flexible to allow relaxing the requirements in selecting nodes from t_i.

Example 6.4.1. *Consider the following transformation:*

$$\texttt{if}(\mathit{chunk\_end} + *\mathit{ch} < \mathit{skb})\ \{...\} \mapsto$$
$$\texttt{if}(\mathit{chunk\_end} + *\mathit{ch} <= \mathit{skb})\ \{...\}$$

The right-hand side of the corresponding partial transformation rule is:

$$\begin{aligned}&\texttt{Tree}(\texttt{IfStatement}, [], [\\ &\quad\texttt{Tree}(\texttt{BooleanExpression}, [], [\\ &\qquad \mathit{select}_1, \texttt{Tree}(\texttt{Opcode}, [\texttt{"<="}], []), \mathit{select}_2])\ ])\end{aligned}$$

where select_1 is specified by the `guard`*:*

$$\begin{aligned}&\texttt{HasType}(\texttt{node}, \texttt{Integer}) \sim \mathit{false} \wedge\\ &\texttt{IsKind}(\texttt{node}, \texttt{BinaryOperator}) \sim \mathit{true} \wedge\\ &\texttt{IsKind}(\texttt{node.kids}[2], \texttt{DeReferExpr}) \sim \mathit{true} \wedge\\ &\texttt{IsKind}(\texttt{node.kids}[2].\texttt{kids}[1], \texttt{Identifier}) \sim \mathit{true} \wedge\\ &\texttt{Attribute}(\texttt{node.kids}[2].\texttt{kids}[1], \mathit{Code}) = \texttt{"}\mathit{ch}\texttt{"} \sim \mathit{true} \wedge \ldots\end{aligned}$$

and select$_2$ is specified by the guard

$$\texttt{HasType}(\texttt{node}, \texttt{Integer}) \sim \textit{false} \wedge$$
$$\texttt{Attribute}(\texttt{node}, \textit{Code}) = \text{"}\textit{skb}\text{"} \sim \ \textit{true}$$

Flexible predicates allow select operations to be relaxed. For example, a relaxed select$_1$ allows for a different "**Code**" *to be used.*

By default, FixMorph marks predicates over the "Code" (generally, only leaf nodes have a Code attribute.), "FunctionName", "FileName", and "Kind" attributes as flexible, and predicates over "Type" as non-flexible. The intuition is that predicates over node types determine the high-level structure of the transformation, and are more likely to be preserved over different versions of the same code.

Remark 6.4.1. *A predicate that is marked as flexible is not necessarily relaxed by the synthesis process. Relaxing all flexible predicates will produce an over-generalized transformation rule, which may produce false positives. For instance, an over-generalized R^p from Example 6.4.1 may incorrectly transform an unrelated node, e.g.,* $\texttt{if}(\mathtt{a} + \mathtt{b} < \mathtt{c})\{\dots\}$.

6.4.4 Relaxing a Transformation Rule

Once a partial transformation rule R^p is synthesized for $\mathtt{v_{mainline}}$ patch p, FixMorph decides how to relax R^p for each old stable version $\{\mathtt{v_1}, \dots \mathtt{v_n}\}$. To help with this process, we introduce the notion of *alignment* between different versions.

Alignment We define an *alignment* to be a set of mappings between the code elements or context of $\mathtt{v_{mainline}}$ and each $\mathtt{v_i}$. For example, given the following expressions:

$$\begin{array}{ll} \texttt{skb\_pull}(\textit{skb}, *\textit{ch}) & \text{from } \mathtt{v_{mainline}} \\ \texttt{skb\_pull}(\textit{skb}, \textit{sctp\_chunkhdr\_t}) & \text{from } \mathtt{v_i} \end{array}$$

an alignment of $\mathtt{v_{mainline}}$ and $\mathtt{v_i}$ would be $\{\textit{skb} \mapsto \textit{skb}, *\textit{ch} \mapsto \textit{sctp\_chunkhdr\_t}, \dots\}$. From the alignment FixMorph builds multiple mappings, including:

- File: maps of the files between $\mathtt{v_{mainline}}$ and $\mathtt{v_i}$;

- Function: maps of functions between matched file pairs;

- `Expression`: maps of the matched expressions, e.g., $*ch \mapsto sctp\_chunkhdr\_t$;
- `NameSpace`: maps of the matched identifiers.

First, FixMorph aligns the source files from $\mathtt{v_{mainline}}$ to $\mathtt{v_i}$ using a combination of the Git version control history and clone detection. For each modified file, FixMorph uses git to determine the name of the corresponding file in the target version. If git produces no information, FixMorph uses clone detection [55] to find the file in the target version that is most similar to the modified file in the mainline. Next, FixMorph aligns each function, expression, and namespace in the affected files using a combination of GumTree [34] and anti-unification [109]. Given two ASTs t_1 and t_2, GumTree can generate an *edit script* comprised of `insert`, `delete`, `move` and `update` operations that can transform t_1 into t_2. Besides, GumTree also constructs a set of matched pairs for the unchanged code elements. For our application, we re-purpose GumTree to generate mappings between two ASTs rather than generate an edit script. Specifically, the GumTree `update` operation can be used to derive a set of maps between the code elements (e.g. member accesses, variables) between $\mathtt{v_{mainline}}$ and $\mathtt{v_i}$.

Using GumTree, the mappings between the same kinds of code elements (e.g., identifier to identifier or, assignment to another assignment), can be extracted. We then use an approach based on anti-unification [109] to generate other kinds of mappings such as expression to identifier (e.g., $*ch$ to $sctp\_chunkhdr\_t$). To do so, we analyze the alignment between the matched non-leaf pairs of $\mathtt{v_{mainline}}$ and $\mathtt{v_i}$ via anti-unification. Given ASTs t_i and t_o, their anti-unification is given by $(\tau, \langle\sigma_1, \sigma_2\rangle)$, where τ is an AST with labelled holes $\{h_0, \ldots, h_n\}$, and two substitutions $\sigma_1, \sigma_2 : \{h_0, \ldots, h_n\} \rightarrow \mathtt{nodes}$ such that $\sigma_1(\tau) = t_i \wedge \sigma_2(\tau) = t_o$. We then use the substitutions to generate a mapping $\sigma_1^{-1}\sigma_2$ between the nodes of t_i and t_o. The mappings produced by GumTree and anti-unification are combined to produce the complete mapping.

Example 6.4.2. *Given the following if-statements:*

$$\mathtt{if}(chunk\_end + *ch < skb)\ \{...\}$$

$$\mathtt{if}(chunk\_end + sctp\_chunkhdr\_r < skb)\ \{...\}$$

We can apply anti-unification to their ASTs to generate: ($\texttt{if}(chunk\_end + h_1 < skb)$, $\langle h_1 \mapsto *ch, h_1 \mapsto sctp\_chunkhdr\_r\rangle$). *The anti-unification result is then used to derive the mapping* $\{*ch \mapsto sctp\_chunkhdr\_r\}$.

Relaxation Once FixMorph generates a map $\{\texttt{node}^{\texttt{i}}_{\texttt{1}} \mapsto \texttt{node}^{\texttt{o}}_{\texttt{1}}, \ldots, \texttt{node}^{\texttt{i}}_{\texttt{m}} \mapsto \texttt{node}^{\texttt{o}}_{\texttt{m}}\}$ between $\texttt{v}_{\texttt{mainline}}$ and $\texttt{v}_{\texttt{i}}$, FixMorph relaxes R^p as follows. Suppose a flexible predicate is presented as $\texttt{pred}(\texttt{node}, \texttt{property})$, meaning a predicate on the `property` (e.g. "`Type`", "`Kind`", "`Code`", etc.) of `node`. FixMorph relaxes such a predicate if and only if the property of `node` is different from the property of its mapped node. FixMorph relaxes `pred` and all the predicates on `node`'s children.

Example 6.4.3. *Let us revisit Examples 6.4.1 and 6.4.2. For the predicate observed in the example,* $\texttt{IsKind}(\texttt{node.kids}[2], \texttt{DeReferExpr})$ *in* $select_1$*, its corresponding node from* $\texttt{v}_{\texttt{mainline}}$ *is* $*ch$*, while the mapped node from* $\texttt{v}_{3.5}$ *is* $sctp\_chunkhdr\_r$*. Since the* ***Kind*** *of* $*ch$ *is different from the* ***Kind*** *of its mapped node, FixMorph relaxes this flexible predicate. Besides, FixMorph relaxes the predicate on* $*ch$*'s child nodes, including:*

$$\texttt{IsKind}(\texttt{node.kids}[2].\texttt{kids}[1], \texttt{Identifier}) \quad \textit{and}$$
$$\texttt{Attribute}(\texttt{node.kids}[2].\texttt{kids}[1], \textit{Code}) = \texttt{"}ch\texttt{"}$$

With the relaxed $select_1$*, the transformation rule can generate the transformation:*

$$\texttt{if}(chunk\_end + sctp\_chunkhdr\_r \texttt{ < } skb)\ \{...\} \mapsto$$
$$\texttt{if}(chunk\_end + sctp\_chunkhdr\_r \texttt{ <= } skb)\ \{...\}$$

6.4.5 Applying the Transformation Rule

Applying the learnt rule to $\texttt{v}_{\texttt{i}}$ itself may not be adequate to successfully transform the program. Although FixMorph learns the transformation rule, the transformed AST could still be incomplete. To make it complete, FixMorph may make a set of post-processing changes, as articulated in the following:

Add missing dependencies The backported patch p_i may depend on some variables, functions, arguments, etc. that are missing in `vi`. FixMorph detects such missing dependencies used by p_i and rectifies them by importing such dependencies. Specifically, FixMorph analyses the AST nodes that are referenced by p_i to find references to missing variables, functions, macros, etc. FixMorph then recursively adds the missing definitions (such as a function or variable declaration, header, etc) to $\mathtt{v_i}$.

Prune irrelevant transformation Pruning of irrelevant transformations may be required to apply the transformation rule to version $\mathtt{v_i}$. The commit that introduced patch p to $v_{\mathtt{mainline}}$ may include some version-specific changes that cannot be backported to $\mathtt{v_i}$. For instance, the commit in $\mathtt{v_{mainline}}$ may move a code statement from one location to another location, whereby the code statement does not exist in $\mathtt{v_i}$. FixMorph detects transformations that should be pruned by treating each change introduced by patch p separately. If FixMorph fails to find an alignment in $\mathtt{v_i}$ for a modified statement, FixMorph prunes the corresponding transformation.

Patch Validation FixMorph applies R with the above mentioned adjustments from post-processing, to backport the patch p from $\mathtt{v_{mainline}}$ to $\mathtt{v_i}$. FixMorph first validates the patched $\mathtt{v_i}$ via compilation to check for build errors. If tests are available, FixMorph can further validate the patched $\mathtt{v_i}$.

6.5 Implementation

Although we synthesize transformation rules using a Refazer-like approach, we cannot directly reuse the Refazer tool since it is designed for C# and Python programs. FixMorph is composed of three main components (*Build engine*, *Transformation rule synthesis* and *Source code transformation*) and amounts to 10,918 code lines in Python and 2,645 code lines in C++.

The *Build engine* is used initially to build typed ASTs and finally to validate the patched code. The build engine is based on LLVM/Clang, to benefit from its facilities for source-to-source transformation and handling of macros. Clang does, however, elide `#ifdef`s, which can lead to missing some code. To limit the number of cases that are considered, our build engine tries two strategies 1) rewrite all `#ifdef`s to `#if 1` and 2) rewrite all `#ifdef`s to `#if 0`.

Transformation Rule Synthesis. To synthesize transformation rules, we used Clang to translate the concrete patch to the extended DSL. To generate alignment, we use the LLVM GumTree implementation as the AST differencing algorithm [75]. The ASTs used by the original LLVM GumTree implementation only include `NodeKind` and `Code`; we added information about types, position, function names, filenames, etc.

Source code Transformation. While our synthesis algorithm is expressed in terms of ASTs, FixMorph transforms source code by leveraging the unique source to source transformation features provided by Clang/LLVM. Accordingly, the code layout and comments not affected by the patch are preserved.

6.6 Evaluation

In this section, we evaluate the effectiveness of FixMorph in backporting patches and answer the following research questions:

RQ1 Can FixMorph backport fixes of security vulnerabilities?

RQ2 How effective is FixMorph in backporting patches?

RQ3 How does FixMorph compare with existing tools?

6.6.1 Experimental Setup

Dataset: To evaluate FixMorph, we build our dataset in the form of patch pairs $(\mathtt{p}_{\mathtt{mainline}}, \mathtt{p}_{\mathtt{i}})$, where $\mathtt{p}_{\mathtt{mainline}}$ is the patch committed to the mainline version, and $\mathtt{p}_{\mathtt{i}}$ is the patch backported to $\mathtt{v}_{\mathtt{i}}$. We build our dataset according to the following criteria:

- Patch $\mathtt{p}_{\mathtt{mainline}}$ was submitted to the mainline during 2011-2019 and versions below 5.0;
- To generate typed ASTs, the mainline version should be compilable before and after introducing $\mathtt{p}_{\mathtt{mainline}}$, and version $\mathtt{v}_{\mathtt{i}}$ should be compilable before $\mathtt{p}_{\mathtt{i}}$. We omit the subjects for which we cannot generate complete ASTs;

- Our prototype only supports modification to *.c files, not header files, hence the patch should only modify *.c files. Further to reduce the complexity, we select patches affecting a single *.c file.

- If $\mathtt{p_{mainline}}$ has been backported to multiple versions, we select the oldest one as $\mathtt{p_i}$ which represents the most challenging task;

- We eliminate the patches that have been used in our study (Section 6.2.3) to ensure no overlap between our study and evaluation.

Selecting patches affecting only a single *.c file may indeed focus the evaluation on simpler patches. Nevertheless, we find that 80% of all backported patches in the Linux kernel (42036/51663) affect only a single file. We filter the backported patches using the above criteria, and randomly select 350 pairs to construct our dataset. Table 6.2 shows the distribution of the patch size (number of lines changed) in our dataset.

Table 6.2: Patch size distribution in FixMorph dataset

Lines	1-2	3-4	5-6	7-8	9+	Total
Patches	165(47%)	78(22%)	50(14.5%)	29(8.5%)	28(8%)	350(100%)

Moreover, we evaluate FixMorph in backporting security vulnerabilities by selecting 30 patches that fix CVEs using the same criteria. We focus on the CVEs reported during 2014-2019, and made sure that the 30 CVE patches are disjoint from the 350 patches in our main dataset.

All experiments are conducted on a Dell PowerEdge R530 with Intel(R) Xeon(R) CPU E5-2660 processor and 64GB RAM.

6.6.2 Experimental Results

6.6.2.1 Backporting Security Vulnerability Fixes

To investigate the usefulness of FixMorph in backporting security vulnerability fixes, we evaluate FixMorph on 30 CVE fixes. Table 6.3 shows the statistics of our targeted CVEs, including the CVE id, vulnerability type, the patch commit id, and the release and target versions. It also shows the evaluation results, where the column“Result” indicates whether the backported patch is semantically equivalent to the developer backported patch.

FixMorph was able to successfully backport 21 out of 30 CVE patches fixing a variety of bugs with semantic equivalence to the developer ported patch. These results suggest that FixMorph can be useful in helping developers fix security vulnerabilities effectively. We also manually analyzed the reasons for the failed cases. For some cases (*e.g.*, CVE-2018-10879), FixMorph could not determine the correct patch locations because the mainline and target version are very different. For some cases, the adaptation requires complex code changes that would involve understanding the patch semantics.

6.6.2.2 Effectiveness of FixMorph

To evaluate the effectiveness of FixMorph, for each pair ($\mathtt{p}_{\mathtt{mainline}}$, $\mathtt{p}_{\mathtt{i}}$), we use FixMorph to automatically backport $\mathtt{p}_{\mathtt{mainline}}$ from the mainline to $\mathtt{v}_{\mathtt{i}}$, and use the developer backported patch $\mathtt{p}_{\mathtt{i}}$ to verify the correctness of the auto-backported patch. We evaluate the correctness of the auto-backported patches by checking their syntactic and semantic equivalence with the developer backported patches.

Table 6.4 summarizes our evaluation results. Column “Type” indicates the class of subjects as defined in Section 6.2.3 and “Total” is the number of pairs for each type. Column “Plausible” shows the number of backported patches that can be compiled in the form of x (y%), where x is the total number of instances that were backported and y represents the percentage. Columns “Syntactic” and “Semantic” represent the number of patches that are syntactically and semantically equivalent to the developer backported patch, respectively. Out of the 350 subjects, FixMorph can backport 285 of them without introducing build failures, which accounts for 81.4%. 245 subjects (70.0%) result in code that is identical to the developer’s patch, while 263 subjects (75.1%) result in code that is semantically equivalent to the developer’s backported patch. FixMorph shows good results in Type-I, II, and III, indicating its effectiveness in identifying correct patch locations and changing the namespace. Type-IV requires changing both the patch location and namespace, which is more challenging, but FixMorph still can correctly backport 54.7% of them. Type-V includes the most challenging cases, where 42.7% of the patches are correct. The main reason is that FixMorph fails to transform some

Table 6.3: Results of backporting CVE tagged bug fixes

CVE ID	Vuln Type	Patch Commit	Release Version	Target Version	Result
CVE-2018-1118	IL	670ae9ca	4.17	4.9	✓
CVE-2018-19985	MO	5146f95d	4.20	3.16	✗
CVE-2019-3701	DoS	0aaa8137	5.0	3.16	✓
CVE-2017-0786	IL	17df6453	4.14	3.16	✓
CVE-2018-1092	NPD	8e4b5eae	4.16	3.2	✓
CVE-2018-1108	RNW	dc12baac	4.17	4.14	✗
CVE-2014-8481	NPD	a430c916	3.18	3.17	✓
CVE-2015-7513	DZ	0185604c	4.4	3.2	✓
CVE-2018-16658	IL	e4f3aa2e	4.19	3.16	✓
CVE-2018-1094	NPD	a45403b5	4.16	4.14	✗
CVE-2018-9363	IO	7992c188	4.18	3.16	✓
CVE-2018-10881	MO	6e8ab72a	4.17	3.16	✓
CVE-2018-10879	UAE	5369a762	4.17	3.16	✗
CVE-2016-9191	DoS	93362fa4	4.10	3.12	✓
CVE-2018-10880	DoS	8cdb5240	4.17	3.16	✗
CVE-2016-0728	IO	23567fd0	4.4	3.10	✓
CVE-2018-11412	MO	117166ef	4.17	3.16	✓
CVE-2017-7184	MO	677e806d	4.11	3.2	✓
CVE-2015-5257	NPD	cbb4be65	4.3	3.2	✗
CVE-2017-12153	NPD	e785fa0a	4.14	3.2	✓
CVE-2016-0758	IO	23c8a812	4.6	3.12	✓
CVE-2016-6213	DoS	296990de	4.12	4.1	✓
CVE-2014-9529	MO	a3a87844	3.19	3.2	✓
CVE-2017-11600	MO	7bab0963	4.13	3.2	✓
CVE-2017-12193	NPD	ea678998	4.14	3.16	✗
CVE-2016-3713	IL	9842df62	4.6	4.4	✓
CVE-2017-8824	UAF	67f93df7	4.16	3.2	✓
CVE-2016-8650	MO	f5527fff	4.17	3.16	✓
CVE-2017-2584	IL	129a72a0	4.10	3.10	✗
CVE-2018-14633	MO	18164943	4.19	3.16	✗
Total	-	30	-	-	21

RNW: Random Number Weakness, *NPD*: Null Pointer, *DoS*: Denial of Service, *UAF*: Use After Free, *MO*: Memory Overflow, *IL*: Information Leakage, *IO*: Integer Overflow, *DZ*: Divide by Zero

Table 6.4: Effectiveness of FixMorph in backporting kernel patches

Type	Total	Plausible	Syntactic	Semantic
I	1	1 (100%)	1 (100%)	1 (100%)
II	235	216 (91.9%)	204 (86.8%)	204 (86.8%)
III	9	7 (77.8%)	4 (44.4%)	7 (77.8%)
IV	30	22 (73.3%)	16 (53.3%)	19 (63.3%)
V	75	41 (54.7%)	22 (29.3%)	32 (42.7%)
Total	350	285 (81.4%)	245 (70.0%)	263 (75.1%)

complex logic and structural changes. To backport, reasoning about the semantics of those patches is needed, which is out of the scope of this work.

6.6.2.3 Comparison with Existing Tools

To compare FixMorph with existing techniques, we consider the following baseline approaches:

- `patch`: the `patch` tool [72] from GNU Diffutils; by default, `patch` requires the change to occur at the indicated line numbers;
- `patch`^{c}: the `patch` tool in context mode [72] (`-context` option), providing flexibility about how many of the patch's context lines are required to be matched;
- Sydit*: our reimplementation of Sydit [94] for C; Sydit is a program transformation tool for Java that learns a transformation rule from a single example, in which it simply generalizes all the identifiers and patch locations. Sydit* follows Sydit, but uses GumTree [34] instead of ChangeDistiller [35] as the AST differencing algorithm. This should benefit Sydit* because GumTree has been shown to be more accurate than ChangeDistiller.

For a fair comparison, we provide the correct file to patch to all these tools by querying the Git version control system.

Table 6.5 summarizes our quantitative comparison results. Columns 3–6 represent the correctly backported patches by `patch`, `patch`^{c}, Sydit*, and FixMorph, respectively. The result for each tool and each class is shown in the form x (y%), where x is the number of patches that have been correctly backported, and y is the accuracy. Despite having the extra advantage of localizing the correct source file, the `patch` tool still failed to correctly backport around half of the instances

Table 6.5: Quantitative comparison of FixMorph with existing tools

Type	Total	`patch`	`patch`^{c}	Sydit*	FixMorph
I	1	1 (100%)	1 (100%)	0 (0%)	1 (100%)
II	235	124 (53%)	182 (77%)	89 (38%)	204 (87%)
III	9	0 (0%)	0 (0%)	2 (22%)	7 (78%)
IV	30	0 (0%)	0 (0%)	6 (20%)	19 (63%)
V	75	0 (0%)	0 (0%)	0 (0%)	32 (43%)
Total	350	125 (36%)	183 (52%)	97 (28%)	263 (75%)

in Type-II. This illustrates the difficulty in identifying the correct patch locations. In contrast, `patch`^{c} performs better than the `patch` tool because it uses context information to find the correct patch location. The key insight we draw from this observation is that *context information is important in identifying patch locations.* Sydit* performs quite well in backporting patches to the correct location due to the usage of additional AST context information. However, since transformation rules synthesized by Sydit* are usually over-generalized, Sydit* incorrectly backports the patches to many locations where the patch should not be applied. We regard a backport as a *false positive* if it produces a patch that modifies the wrong locations in $\texttt{v}_\texttt{i}$. Overall, Sydit* produces 97 correct patches that are semantically equivalent to the developer patches. FixMorph outperforms all the above tools, especially for the challenging cases, i.e., Type-III, IV, and V. The transformation guided by the alignment of the mainline and target version allows FixMorph to correctly backport more Type-III, IV, and V patches.

Table 6.6: Qualitative comparison of FixMorph with existing tools

Type	`patch`		`patch`^{c}		Sydit*		FixMorph	
	P%	R%	P%	R%	P%	R%	P%	R%
I	100	100	100	100	0	0	100	100
II	77	63	99	78	46	69	95	91
III	0	0	0	0	29	50	100	78
IV	0	0	0	0	38	30	86	70
V	0	0	0	0	0	0	78	48
Total	71	42	82	59	44	43	92	80

To better understand the reliability of each tool, we further evaluate the quality of the transformations for each tool by calculating the precision and recall. Table 6.6 shows the qualitative comparison results. Columns "P%" and "R%" indicate the precision and recall, respectively. Overall, FixMorph produces much fewer incorrectly backported patches (higher precision) and misses much fewer cases that should be patched (higher recall) than the other tools.

6.6.3 Threats to Validity

Several threats may affect the validity of our evaluation. First, since the baseline tool Sydit is designed for Java programs, to compare with it, we implemented Sydit* by ourselves. We tried our best to follow Sydit's design, but the differences in implementation details may still affect its results. Second, although FixMorph shows strong efficacy on the evaluated benchmark, it may perform differently on other subjects. To mitigate this problem, we evaluated FixMorph on a fairly large dataset that covers different scenarios. Last, we manually compare the backported patches with developers' patches to verify their correctness. To reduce the potential bias caused by manual analysis, two authors of this paper independently double checked the correctness of generated patches.

***Limitations of* FixMorph** Our implementation is based on LLVM/Clang, and thus inherits the limitations of that framework. Since handling all combinations of compilation options is not scalable, when compiling the project, we only consider two sets of compilation options (see Section 6.5). This strategy works for most cases, but in some cases, it could result in certain un-compiled blocks of code being unavailable to FixMorph, thus leading to incomplete backporting or even failure. To alleviate this limitation, we allow users to specify the values of preprocessor variables according to their working environment.

6.7 Summary

We investigated the backporting activities in the Linux kernel because it is a large-scale widely used codebase. The sheer complexity of the patches, the diversity of the transformations involved, and the absence of test cases as specification pose additional challenges in fixing security vulnerabilities. Due to the popularity and importance of the Linux kernel, there is a significant practical value for reducing exposure to security vulnerabilities. With the attack surface moving to edge devices (which may be running older versions of Linux), propagating patches to old Linux versions can be a meaningful security enhancement aid.

To reduce the exposure to known vulnerabilities, we study the problem of automated patch backporting, to automatically backport security patches. We propose a technique inspired from program synthesis technique, and our evaluation shows that our implementation FixMorph is effective in backporting security patches (i.e., successfully backported 21/30, that is, 70% of the evaluated CVEs). Although we only evaluate on the Linux kernel, Table 6.3 shows that FixMorph can backport patches for various types of vulnerabilities. Instead of repetitive generation of patches from scratch, our proposed solution helps backport patches to fix software security vulnerabilities that exist in older versions of the same software.

Chapter 7

Vulnerability Repair via Patch Transplantation

In this chapter, we formulate and study a problem related to security vulnerability repair, namely automated patch transplantation. A patch for an error in a donor program is automatically adapted and inserted into a "similar" target program. We observe that despite standard procedures for vulnerability disclosures and publishing of patches, many un-patched occurrences remain in the wild. One of the main reasons is the fact that various implementations of the same functionality may exist and, hence, published patches need to be modified and adapted. Therefore, we propose and implement a workflow for transplanting patches. This chapter introduces the *patch transplantation problem*, and explains why it is an important problem to tackle in the context of repairing software security vulnerabilities. The chapter starts with an overview of the *patch transplantation problem* and its necessity. It continues with an illustrative example to demonstrate the challenges, and formally define the *patch transplantation problem.* Afterwards, the chapter continues with the technical details of our proposed solution and concludes with a preliminary evaluation, showing the effectiveness of our proposed approach to remedy a class of security vulnerabilities known as recurring-vulnerabilities.

7.1 Overview

If the patch for an error or a vulnerability in a buggy program P is available (*e.g.*, the vulnerability has been patched manually and the fix is available), can the patch be automatically transplanted or adapted into another "similar" buggy program P'? We call this the *automated patch transplantation* problem. Many use-cases which can benefit from a solution to automate patch transplantation exist. First, security fixes in the latest software version may be “backported" to older program versions. Such backporting is not restricted to security fixes but also can be used to enhance compatibility issues in software versions, such as managing the collateral evolution of device drivers to enable their functioning despite evolution of the operating system. (*e.g.,* prior work on evolution of Linux backporting [102]).

Second, patch transplantation can be useful for propagating fixes to different implementations of the same protocol or functionality, as opposed to different versions of the same program. Implementations of the same protocol or functionality can differ due to the difference in the programming language or difference in implementation while using the same programming language. Porting a patch across languages is more challenging and beyond the scope of the patch transplantation problem. To elaborate on the transplantation across different implementations, let us consider the *Heartbleed* vulnerability (CVE-2014-0160), which could lead to disclosure of private information by applications using OpenSSL [32]. Although a patch for the Heartbleed vulnerability is available, it cannot be immediately inserted into any OpenSSL implementation, instead the patch needs to be *adapted.* As different web servers rely on different implementations of OpenSSL, Heartbleed continues to persist in the wild [124], despite the patch being widely available. Thus by automatically adapting patches of Heartbleed to other vulnerable OpenSSL implementations, we can reduce the exposure to vulnerabilities.

In general, one of the crucial steps towards defense against published exploits is to integrate available patches into one's system as quickly as possible. The challenge in incorporating patches from different sources is to be able to adapt the code modifications involved. Often, shared libraries are customized with new features, different data structures or rewriting previous implementation to match the integrated environment. Hence, directly applying a general patch is not trivial and sometimes difficult.

Problem Statement Given buggy and fixed donor programs P_a, P_b, and a buggy program similar to P_a, also called a host or target program P_c, the goal is to fix P_c to produce a fixed version of P_c, namely P_d. We assume that P_a and P_c fail on the same failing input t_F.

For security patches, t_F is an exploit which takes advantage of an existing software vulnerability. A formulation of the automated patch transplantation problem explaining the inputs and outputs of the problem appears in Figure 7.1. Note that P_a, P_b, P_c, t_F are inputs to the patch transplantation problem, and the output is P_d, the program with the transplanted patch.

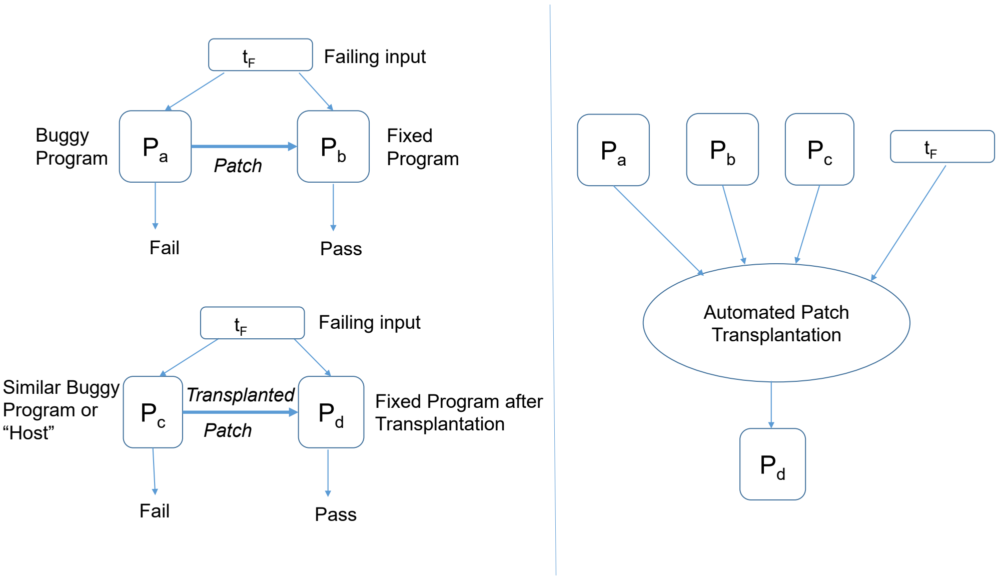


Figure 7.1: The Automated Patch Transplantation Problem

Differences with other problems studied We note that the patch transplantation problem formulated by our work is *different*, though related, to the program repair problem and the program transplantation problem. The program repair problem seeks to (minimally) modify a program so as to meet a correctness criteria such as passing a given test-suite. To relieve the burden of fixing bugs, many techniques have been previously proposed for automated program repair such as genetic programming, semantic analysis based repair techniques, and machine learning guided techniques. Unlike the automated repair problem we do not try to synthesize patches or fit into patch patterns; instead the patch transplantation problem is more "goal-directed"— it automatically identifies a patch from the donor, extracts the identified patch, computes an insertion location for the patch in the target program and inserts that patch by adapting to the context of the insertion point. The patch transplantation problem is also different from the program transplantation problem. The program transplantation problem deals with transplanting a feature from program P into another program P' such that the transplanted feature must not disrupt the existing functionality of its target P' and must actually execute and add the functionality of the desired feature to its target. These techniques are limited to transferring a logical block of code (such as a function or a check) instead of patches that may involve several disjoint blocks of code. Moreover, program transplantation techniques typically require manual identification of the insertion point, unlike our patch transplantation problem where the insertion point is automatically identified.

7.2 Example

We next present an example of an integer overflow error in OpenJPEG (a C library for the open-source JPEG2000 codec) to have a better understanding of the challenges in the patch transplantation problem. Figure 7.2 shows the integer overflow error in OpenJPEG, this code snippet is simplified for brevity. There is a potential overflow at line 560, where OpenJPEG allocates memory to `cp->tcps` by computing the value as `cp->tw * cp->th`. Input image files with large width and height fields may cause the calculation at line 560 to overflow, eventually writing beyond the end of the allocated buffer. In the error-triggering input, the JPG file height field is 210 and the width field is 2147483646.

```
405  static void j2k_read_siz (opj_j2k_t *j2k) {
415   image->y1 = cio_read(cio, 4);   /* Ysiz */
416   image->x0 = cio_read(cio, 4);   /* X0siz */
417   image->y0 = cio_read(cio, 4);   /* Y0siz */
418   cp->tdx = cio_read(cio, 4);     /* XTsiz */
419   cp->tdy = cio_read(cio, 4);     /* YTsiz */
420   cp->tx0 = cio_read(cio, 4);     /* XT0siz */
421   cp->ty0 = cio_read(cio, 4);     /* YT0siz */

423   if ((image->x0<0)||(image->x1<0)||(image->y0<0)||(image->y1<0)) {
424     opj_event_msg(j2k->cinfo, ...);
427     return;
428   } ...
517   cp->tw = int_ceildiv(image->x1 - cp->tx0, cp->tdx);
518   cp->th = int_ceildiv(image->y1 - cp->ty0, cp->tdy);
519   ...
      /* the overflow error */
560   cp->tcps = (opj_tcp_t*) opj_calloc(cp->tw * cp->th, sizeof(opj_tcp_t)); ...
622  }
```

Figure 7.2: Overflow error in OpenJPEG 1.5.1

Automated Program Repair. Consider the scenario where we attempt to fix this bug using two state-of-the-art program repair tools (F1X[89] and Prophet[79]). Since these two APR techniques require a test suite, we created a test suite inclusive of the failing test case and a passing test case, and also provided the correct location for the fix to generate the patch, which enables us to compare the two patches generated at the same location. Figure 7.3a and 7.3b show the two patches generated by F1X and Prophet, respectively. F1X was able to generate a patch which modifies an existing if statement to avoid the failing test case. However, the fix in Figure 7.3a does not generalize for test cases beyond the given test suite, since it only fixes the two given test cases. Similarly, Prophet generated a patch which omits the execution of a statement (i.e., Line 518 in Figure 7.2) by inserting a condition which is always false (i.e., semantically equivalent to deleting a statement). The correct patch for this bug would be to evaluate if the computation of `cp->tw * cp->th` would result in an overflow and avoid the overflow by following an error handling procedure. Both APR generated patches failed to generalize the patch to the extent of avoiding the overflow, rather generated a patch that could simply pass the failing test suite. Prophet is a technique based on machine learning which generates arbitrary code changes and relies on the test suite for correctness, whereas F1X is semantic-based which uses code analysis to generate the patch that attempts to address the underlying bug rather than changing the code just enough to satisfy the test suite. However, it relies on the test suite to generate the correct constraints to obtain the correct patch. If sufficient test cases are provided, the generated patch would be more generalized. This example highlights one of the limitations of current automated program repair techniques, which is generally known as the overfitting problem [133, 110, 87]. As illustrated in our motivational example, APR generated patches cannot repair bugs without sufficient number of test cases to generalize the patch, and the sufficient number differs from one bug to another. When used to fix security vulnerabilities, such inaccurate patches could lead to undesirable effects by believing that the vulnerability has been fixed when in fact it has not.

Patch Transplantation. A different implementation of the JPEG2000 codec can be found in JasPer (a utility for image manipulation) which could also serve as a fix for this vulnerability in OpenJPEG. We discovered that the input file that

```
 static void j2k_read_siz (...) { ....
- if ((image->x0<0)||(image->x1<0)||
    (image->y0<0)||(image->y1<0)) {
+ if ((image->x0==0)||(image->x1<0)||
    (image->y0<0)||(image->y1<0)) {
  ....
}
```

(a) Patch generated by F1X

```
 static void j2k_read_siz (...) { ....

+ if (!(1))
    cp->th = int_ceildiv(image->y1 - cp->ty0,
         cp->tdy);
  ....
}
```

(b) Patch generated by Prophet

Figure 7.3: Patches generated using Automated Program Repair

exploits the vulnerability in OpenJPEG 1.5.1 is able to exploit the same vulnerability in JasPer 1.900.12, and also fixes the bug in JasPer 1.900.13. Our proposed approach will extract the code that fixes the bug in JasPer 1.900.13 and insert into OpenJPEG 1.5.1 in the following way. First, we build the binaries with an integer sanitizer to identify potential overflow locations. Next, we check if the two programs share the same vulnerability. In our example, both JasPer 1.900.12 and OpenJPEG 1.5.1 throw integer overflow errors due to multiplication operation of 210 * -2147483646. We identify the line number jpc_dec.c@1234 in JasPer 1.900.12 as the donor buggy location and line number j2k.c@560 in OpenJPEG 1.5.1 as the target buggy location.

Donor Selection. Figure 7.5a shows the patch for JasPer 1.900.13 which includes a check. Specifically, JasPer at commit b9be3d9 contains the vulnerability and JasPer at commit d91198a contains the fix (i.e., the pair selection is JasPer-b9be3d9 (P_a), JasPer-d91198a (P_b), OpenJPEG-1.5.1 (P_c)).

Patch Extraction. Next, analyses the source code diff and aligns the execution trace of P_a and P_b to narrow down the changes to be the patch that fixes the bug. Lines 1198 and 1235-1238 in Figure 7.5a are the changes in JasPer 1.900.13 that checks if the multiplication results in an overflow through a function `jas_safe_size_mul`. This function determines if an overflow will occur if the two parameters (`dec->numhtiles` and `dec->numvtiles`) are multiplied. If an overflow occurs, it will return false, otherwise, it will return true and assign the multiplication result to the `size` variable.

Concolic Execution. Once we identify the patch for transplantation, we need to translate the statements in the patch and find the insertion point. For this purpose, we perform concolic execution [119] on all three programs in our pair selection with the same input file to capture the symbolic paths for the execution of each program. When identifying the insertion point for the transplantation, the patch can be inserted at any location from the starting point of the execution trace to the crashing point (or a suspicious buggy point). As there could be potentially many candidate locations, we identify a divergent point (see Def 4) in P_c, similar to the divergence caused by the patch in P_b with respect to P_a. A divergence in the trace of P_b with respect to P_a is caused by a code change (i.e. patch) which resulted in the difference and is most likely to be a potential divergent point at which the patch has been applied to. One of the potential divergent points is the condition at line 1235 in Figure 7.5a. We find a similar location in P_c using partial path condition dominance (see Def. 6). We calculate the partial path condition in P_b at line 1235 in Figure 7.5a, and we traverse through the execution trace of P_c to find a similar location where the partial path condition dominates, i.e d_c.

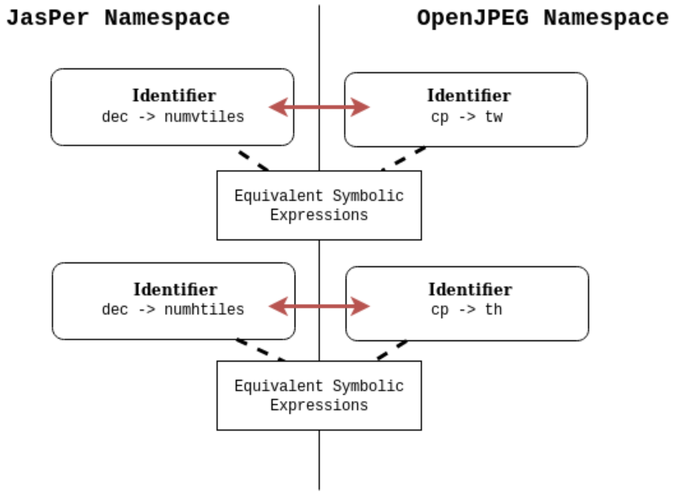


Figure 7.4: Variable mapping between OpenJPEG and JasPer

Candidate Function. Once we have identified a divergent point in P_c, the next step is to traverse through the estimated divergent point d_c and the crashing point l_c in P_c, in order to identify the candidate functions to transplant the patch. For each function f executed in P_c from and inclusive of d_c up to l_c, we consider the variables used within the function f. For each variable v in f, we capture the symbolic expressions and generate a mapping with the variables used in our patch. More precisely, we check for a function that has a mapping for the variables `dec->numvtiles` and `dec->numhtiles`, which is likely to be a candidate function. Among all candidate functions, we choose the first candidate in the trace (that is the function that executes first) for two reasons: (1) the patch can impact more paths and (2) the vulnerability is fixed earlier in the execution of the failing input. In our example, the divergent point and the crashing point lie in the same function `j2k_read_siz`. However, this may not hold for the general cases as the divergent point and the crashing point can be in two different functions.

Candidate Location. We consider the availability of the variables we mapped at previous stage, for each location in our candidate function to find candidate locations for our patch. For each statement inside the function, we compute the list of available variables, and find the candidate points where the variables in our generated mapping are usable. Among the candidate locations, we choose the first candidate for the same reasons mentioned above. In our example, we choose line number 519 in Figure 7.2.

Code Transplantation. Once we have identified the insertion location as line number 519 in Figure 7.2, our next step is to translate the patch to the namespace of P_c, and insert the code at the identified insertion location. We make use of AST node context information from both P_a and P_c programs to adapt to the insertion point context. Then, we translate the variables to the namespace of OpenJPEG using symbolic analysis (explained in Section 4.3), where we obtain the mappings of `dec->numvtiles` into `cp->tw` and `dec->numhtiles` into `cp->th` as illustrated in Figure 7.4. We use this mapping to translate the variable names in the patch while weaving the patch into the insertion point in P_c. Using dependency analysis, we identify that `jas_safe_size_mul` function is missing in our target program P_c. We perform transplantation of this function by following the same steps (i.e., extract the missing source, translate the variables, find a suitable insertion location and transplant into the target program) such that the function can be called within the inserted patch.

Final Result. Patch transplantation is able to successfully repair the bug in OpenJPEG 1.5.1, preventing any potential buffer-overflows due to the integer overflow caused during the calculation of the buffer size as shown in Figure 7.5b. Although F1X [89] and Prophet [79] were able to generate patches that pass the failing test case, the generated patches are of poor quality due to the quality of the test suite. This is where, patch transplantation differs from test driven patch generation: a)

```
1188  static int jpc_dec_process_siz(...)
      {
         ....
1198  + size_t size;
         ...
        /* overflow check */
1235  +   if (!jas_safe_size_mul(dec->numhtiles,
           dec->numvtiles, &size)){
1236  +        return -1;
1237  +   }
      -   dec->numtiles = dec->numhtiles * dec->
           numvtiles;
1238  +   dec->numtiles = size;
1241      if (!(dec->tiles=jas_alloc2(dec->numtiles
              ,sizeof(jpc_dec_tile_t)))){

1242          return -1;

1243      }
          ....
1290  }
```

(a) Developer patch in JasPer 1.900.13

```
     /* Adapted patch for OPENJPEG 1.5.1 */
48   + #define SIZE_MAX    (18446744073709551615UL)
56   + inline static bool jas_safe_size_mul(size_t
          x, size_t y, size_t *result){
57   + if (x && y > SIZE_MAX / x) {
58   +   return false;
59   +   }
60   + *result = x * y;
61   +   return true;
62   + }

440   static void j2k_read_siz (opj_j2k_t *j2k) {
554  + size_t size;
556  + if (!jas_safe_size_mul(cp->tw, cp->th, &size
          )) {
557  +    return -1;
558  + }
       /* the overflow error */
601      cp->tcps = (opj_tcp_t*) opj_calloc(cp->tw *
              cp->th, sizeof(opj_tcp_t));
663  }
```

(b) Transplanted Patch in OpenJPEG 1.5.1

Figure 7.5: Patch generated using transplantation

the patch is a human-written patch which is extracted and then adapted to the context of the target program (i.e., P_c). Since human-written patches are more reliable and general than the generated patches via APR, we eliminate the problem of overfitting as further proved in our experimental results in Table 7.6 (Section 7.6). The result of differential fuzzing highlights that the patches generated using APR do not generalize for unseen test cases; b) patch transplantation can fix more bugs compared to program repair techniques because APR relies on a good test suite for fault localization to identify patch locations. Since patch transplantation does not depend on a test suite, with the use of the partial path condition dominance relationship (see Def. 6) we can find the correct patch location.

7.3 Problem Formulation

Although programs with vulnerabilities may not share common code, they can share different implementations of the same protocol (e.g., OpenSSL) or same standard (e.g., JPEG 2000). Hence, finding a vulnerability in one program can lead to malicious users adapting attacks to other similar programs. This is the scenario we seek to prevent via automated patch transplantation. In this section, we first introduce the notations that we will use in this chapter and then formulate the problem of patch transplantation. P_a represents the buggy version of a program, whereas P_b (also known as the *donor* in the terminology used in software transplantation [9]) denotes the subsequent version in which the fault in P_a is fixed. t_F represents the test that failed in P_a but passes in P_b, while P_c denotes the target program (also known as the *host* in the terminology used in software transplantation [9]) that fails in t_F.

Definition 1 (Similar Vulnerability). *We consider two vulnerabilities as similar if there exists a failing test t_F exploiting both vulnerabilities and the two vulnerabilities exhibits the same output in terms of the return code and crashing/buggy instruction. For instance, the motivational example in Section 7.2 where both JasPer 1.900.12 and OpenJPEG 1.5.1 exhibited similar integer overflow vulnerability for the same test case.*

Definition 2 (Similar Programs). *We consider two programs P_a and P_c as similar if there exists a failing test t_F exploiting a similar vulnerability in both programs. Our goal is to transplant a fix of $P_a$into the other similar program P_c.*

Definition 3 (Patch Transplantation). *Given a pair of buggy and fixed programs (P_a , P_b) and a program P_c similar to P_a, we try to extract the patch between P_a and P_b. The patch is then inserted into P_c which involves finding an insertion point, and adapting the patch with new variable mappings and context information.*

Type	missing dependency?	namespace translation?	example
Class-I	No	No	porting across forks
Class-II	Yes	No	backporting
Class-III	No	Yes	collateral evolution
Class-IV	Yes	Yes	collateral evolution

Table 7.1: Classes of patch transplantation

7.3.1 Classes of Patch Transplantation

There are three major challenges in transplanting a patch from P_b to P_c due to the differences in the two programs. The first challenge is the difference between the namespace and data structures used in the two implementations where the identifiers are not identical, hence an adaptation for the variables is required. The second challenge is to identify and transplant missing dependencies for the patch to work. For example, the patch would require a supplementary function, a subroutine or a definition that is used in the patch, which is missing in the target program P_c. The third challenge is to correctly identify the insertion location of the patch in P_c.

To perform a thorough analysis of the patches, we identify four classes of patches based on the origin of the patch and the adaptation required for the target system to apply the patch (Table 7.1). Further, we define an equivalence relation between the original patch and transplanted patch based on the adaptation required as given below.

Syntactically Equivalent. $Patch_{fix}$ is "Syntactically Equivalent" if $Patch_{orig}$ and $Patch_{fix}$ are exactly the same code. If the namespace and data structure of the two programs P_b and P_c are identical, and the code is identical the transplanted patch would be syntactically equivalent to the original patch.

Semantically Equivalent. $Patch_{fix}$ is "Semantically Equivalent" if $Patch_{orig}$ and $Patch_{fix}$ are not syntactically the same but produce the same semantic behavior. This requires a namespace and/or data structure translation.

7.3.1.1 Class-I : Syntactically Equivalent Transplantation

No adaptation is required for the transplantation. This is the trivial case where the original patch can be applied directly. An example is the backporting of patches to past versions of the same program or porting patches across forked projects.

The patch for CVE-2018-14526 is an example for Class-I. It is a vulnerability in the processing of EAPOL-Keyframes for wpa_supplicant[1]. An attacker could modify the frame to bypass authentication. To fix this vulnerability, an official patch[2] in Listing 7.6a was released and adapted by every operating system that provides the wpa_supplicant driver. FreeBSD driver had to integrate this patch to two different versions of its forks and Listing 7.6b shows the patch[3] for FreeBSD

[1] wpa_supplicant is a WPA Supplicant for Linux, BSD, Mac OS X, and Windows with support for WPA and WPA2 (IEEE 802.11i /RSN)

[2] https://w1.fi/security/2018-1/0001-WPA-Ignore-unauthenticated-encrypted-EAPOL-Key-data.patch

[3] https://www.freebsd.org/security/patches/SA-18:11/hostapd-10.patch

```
int wpa_sm_rx_eapol(...){
  ....
  if ((sm->proto == WPA_PROTO_RSN ||
      sm->proto == WPA_PROTO_OSEN) &&
      (key_info & WPA_KEY_INFO_ENCR_KEY_DATA)
            &&
      mic_len) {
+  if (!(key_info & WPA_KEY_INFO_MIC)) {
+     wpa_msg(sm->ctx->msg_ctx, MSG_WARNING,
     "WPA: Ignore EAPOL-Key with encry..");
+   goto out;
+  }

   if (wpa_supplicant_decrypt_key_data(sm, key
        , mic_len, ver, key_data,&key_data_len
        ))
   ....
}
```

(a) Dev patch for wpa_supplicant 2018-1

```
int wpa_sm_rx_eapol(...){
   ....

   if (sm->proto == WPA_PROTO_RSN &&
       (key_info & WPA_KEY_INFO_ENCR_KEY_DATA)
            ) {

+  if (!(key_info & WPA_KEY_INFO_MIC)) {
+   wpa_msg(sm->ctx->msg_ctx, MSG_WARNING,
     "WPA: Ignore EAPOL-Key with encry..");
+  goto out;
+  }

   if (wpa_supplicant_decrypt_key_data(sm, key
        , ver))

   ....
}
```

(b) Dev patch for FreeBSD SA-18:11

Figure 7.6: Example for Class-I: CVE-2018-14526

10.4. The FreeBSD developer had to identify the insertion point in the FreeBSD driver, which is different from the original patch, but no adaptation was required for the patch code itself. This highlights the fact that even for identical patches finding the insertion point is non-trivial as the predicates in which the patch is inserted in Listing 7.6a and Listing 7.6b are different.

There is a potential issue in the case where WPA2/RSN style of EAPOL-Key construction is used with TKIP negotiated as the pairwise cipher. Hence, a patch was released by the standard organization and adapted by every operating system that provides the wpa_supplicant driver. The developers at FreeBSD had to identify the correct insertion point but no adaptation was required for the patch code itself as depicted above. The context is different with respect to the variable names and additional code in place in FreeBSD as shown in listing 7.6b, compared to the original patch shown in listing 7.6a.

7.3.1.2 Class-II : Syntactically Equivalent Transplantation with Dependency

A dependent function is required for the patch to apply the solution. The dependent component could be from the original patch or could be a missing supplementary code in the recipient program. For example, adding functions for the patch from latest version, which is missing in the old version.

CVE-2006-4806 is an example for Class-II, where the patch requires an depen-

```
char load(ImlibImage * im ..) {

 ....
 im->w = w = cinfo.image_width;
 im->h = h = cinfo.image_height;

 + if (!IMAGE_DIMENSIONS_OK(w, h)){
 +  im->w = im->h = 0;
 +  jpeg_destroy_decompress(&cinfo);
 +  fclose(f);
 +  return 0;

 + }
 ....
}
```

(a) Developer's patch in imlib2 1.4.3

```
+ # define IMAGE_DIMENSIONS_OK(w, h) \
+   ( ((w) > 0) && ((h) > 0) && \
+     ((unsigned long long)(w) * \
+     (unsigned long long)(w) <= \
+     (1ULL << 29) - 1) )

char load(ImlibImage * im ..) {
 ....
 im->w = w = cinfo.image_width;
 im->h = h = cinfo.image_height;
 + if (!IMAGE_DIMENSIONS_OK(w, h)){
 +  return 0;
 + }
 ....
}
```

(b) Adapted patch for imlib2 1.4.0

Figure 7.7: Example for Class-II: CVE-2006-4806

dent function to transplant the patch in the recipient program. It occurs due to a buffer overflow in imlib2 (an image file processing library) which could allow remote attackers to cause a denial of service attack. To fix the vulnerability, developers of imlib2 applied a patch (Figure 7.7a) which includes a supplementary function named `IMAGE_DIMENSIONS_OK` which checks if the provided width and height (`im->w` and `im->h`) are within standard limits of the application to prevent memory overflow. The same vulnerability exist in older version of imlib2, specifically in imlib2 1.4.0 which does not include the definition of the function `IMAGE_DIMENSIONS_OK`, hence the transplantation of the patch (Figure 7.7b) involves the dependency for the patch to correctly fix the vulnerability in imlib2 1.4.0.

7.3.1.3 Class-III: Semantically Equivalent Transplantation

In this class of patches, adaptation is required to apply the transformation into the target system due to syntactic differences. For instance, when two programs are semantically equivalent but syntactically different, the patch needs to be modified before transplanting into the recipient program. This requires a namespace translation between P_b and P_c.

CVE-2013-4231 is an example of Class-III class, where the patch requires an adaptation in terms of namespace translation. It occurs due to an buffer overflow in Libtiff, a library for processing Tagged Image File Format files. A bug in one of the library modules which processes GIF images causes an overflow which can be fixed by inserting a check as shown in Figure 7.8b. Since the maximum LZW bits

```
static void ReadImage(....) {
   ....
   if(!ReadOK(fd, &c, 1)) {
        return;
   }

+  if (c > 12)
+     return;
   ....
}
```

(a) developer patch for LibGD 2.0.34 RC1

```
int readraster(void) {
   ....
   datasize = getc(infile);
+  if (datasize > 12)
+     return 0;
   clear = 1 << datasize;
   eoi = clear + 1;

   ....
}
```

(b) developer patch for Libtiff 4.0.4

Figure 7.8: Example for Type III: CVE-2013-4231

allowed in GIF standard is 12, the patch for the overflow error involves inserting a check condition. This vulnerability also exists in LibGD and ImageMagick libraries which are also image processing software similar to Libtiff. All three programs are vulnerable to the same exploit because they follow the same standard for GIF image processing. The adaptation required for the patch is the namespace mapping from `datasize` in Libtiff to `c` in LibGD and change of return type to match the function return type (Figure 7.8a).

```
bool jas_image_cmpt_domains_same(jas_image_t *
     image)
{
 int cmptno;
 jas_image_cmpt_t *cmpt;
 jas_image_cmpt_t *cmpt0;
 cmpt0 = image->cmpts_[0];
 for(cmptno = 1;
     cmptno < image->numcmpts_;
     ++cmptno){

     cmpt = image->cmpts_[cmptno];
     if (cmpt->tlx_ != cmpt0->tlx_ ||
         cmpt->tly_ != cmpt0->tly_ ||
         cmpt->hstep_ != cmpt0->hstep_ ||
         cmpt->vstep_ != cmpt0->vstep_ ||
         cmpt->width_ != cmpt0->width_ ||
         cmpt->height_ != cmpt0->height_) {
         return 0;
     }}
 return 1;
}
```

(a) Original function in Jasper 1.900.14

```
bool jas_image_cmpt_domains_same(
     opj_tcd_tile_t *t)
{
 int cmptno;
 opj_tcd_tilecomp_t *cmpt;
 opj_tcd_tilecomp_t *cmpt0;
 cmpt0 = &t->comps[0];
 for(cmptno = 1;
     cmptno < t->numcomps;
     ++cmptno){

     cmpt = &t->comps[cmptno];

     if (cmpt->x0 != cmpt0->x0 ||
         cmpt->y0 != cmpt0->y0 ||
         cmpt->x1 != cmpt0->x1 ||
         cmpt->y1 != cmpt0->y1) {
         return 0;
     }}
 return 1;

}
```

(b) Adapted function for OpenJPEG 1.5.1

Figure 7.9: Example for Class-IV: CVE-2016-9389

7.3.1.4 Class-IV : Semantically Equivalent Transplantation with Dependency

For syntactically different yet semantically equal patch which requires a dependency to be transplanted, are considered as Class-IV patches. The dependent component, itself may require adaptation due to namespace differences between the donor P_b and the recipient P_c. This requires a namespace translation between P_b and P_c and data structure translation for the patch to work.

CVE-2016-9389 is an example for Class-IV, where the patch requires a dependency to transplant the patch in the recipient program. CVE-2016-9389 is a buffer overflow vulnerability in JasPer 1.900.13 version and fixed in 1.900.14 which allows remote attackers to cause a denial of service attack. The same vulnerability also exists in OpenJPEG 1.5.1, adapting the patch for OpenJPEG 1.5.1 requires mapping variables across different data structures, specifically mapping `jas_image_t` to `opj_tcd_tile_t` and transplanting the missing function from JasPer to OpenJPEG i.e. `jas_image_cmpt_domains_same`. Listing 7.9a and Listing 7.9b depict the difference between the original function and manually adapted function used in the patch.

7.4 Methodology

The goal of `PatchWeave` is to extract a patch from a given donor program and insert into a target program by computing the patch location and adapting the patch to the context of the target program. First, we will introduce the notations that we will use throughout the rest of the chapter, and then we present an overview of our approach and discuss in detail how each phase in our approach works. We will make use of our motivational example presented earlier in Section 7.2 to guide through each phase.

Symbols and Definitions Table 7.2 summarizes the notations used in this chapter, where P_a is used to identify the donor program before the patch and P_b identifies the donor program with the developer fix. Similarly, P_c is used to identify the target program in which we aim to repair the bug and P_d denotes the patched target program after the transplantation.

We now define a *Divergent Point* which identifies a location in the program which

Table 7.2: Annotations used in the Patch Transplantation Problem

Symbol	Description
P_a	the buggy version of the donor program
P_b	the fixed version of the donor program
P_c	the buggy version of the target program
P_d	the fixed version of the target program
t_F	the test case that failed in P_a but passes in P_b
d_a	a divergent point in P_a
d_c	a divergent point in P_c which is mapped to d_a
l_c	a buggy location in P_c where with an observable error
π_a^F	the execution trace of t_F in P_a
π_b^F	the execution trace of t_F in P_b
π_c^F	the execution trace of t_F in P_c

will be used to compute the insertion location for the transplantation. Figure 7.10 shows the divergent points with respect to the source code that differs for P_a, P_b, P_c.

Definition 4 (Divergent Point)**.** *Given two traces π_a^F and π_b^F in P_a and P_b of a failing input t_F, the set of divergent points between π_a^F and π_b^F are the set of locations where π_b^F starts deviating from π_a^F in terms of instructions executed.*

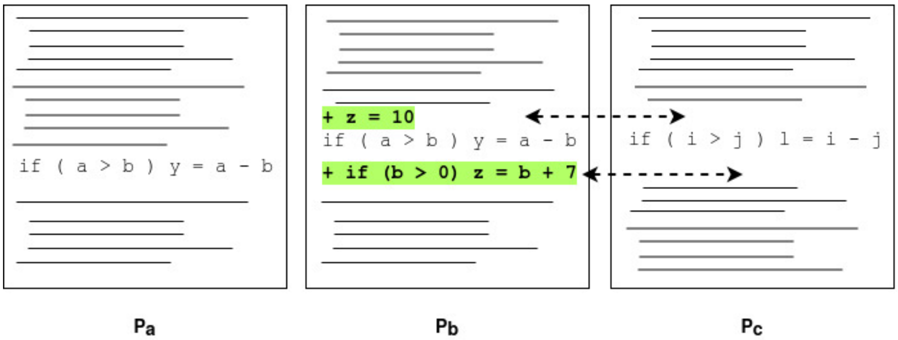


Figure 7.10: Illustration of divergent points for P_a, P_b, P_c

We make use of a relation between two given path conditions π_a and π_b, to map a divergent point from one program to another using the following definitions.

Definition 5 (Partial Path Condition)**.** *Given a trace π_i of an input i in a program P, and given a point l in the trace π_i, the partial path condition of i in program P at l, denoted $ppc(P, i, l)$ is the path condition of π_i up to and including l.*

For the patch transplantation problem that we investigate in this research, we extract the patch from one program and transplant to another similar program (Def 2). Due to the similarity of the two programs, an inherent property is following a standard or a protocol in which the data processing order is the same. For instance, the order of reading/processing input bytes from the input is more or less the same. Making use of this inherent property, we define a relation 'partial path condition dominance' to identify a mapping of program locations from P_a to P_c.

Definition 6 (Partial Path Condition Dominance)**.** *Given two partial path conditions α and γ, we define partial path condition dominance for γ, denoted dom(γ, α), if γ satisfies the condition where input bytes appearing in α are a subset of the input bytes appearing in γ.*

High-level Approach Figure 7.11 shows the overall workflow of `PatchWeave`. Given P_a, P_b, P_c and t_F, we first verify that two programs are similar programs as stated in Definition 2. Then, `PatchWeave` transplants the patch from P_b to P_c in five steps: patch extraction, patch localization, patch adaptation, patch enforcement and patch verification as described in Algorithm 4.

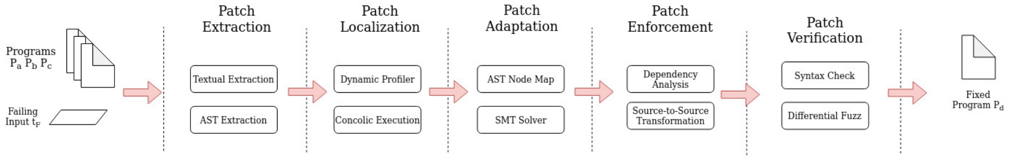


Figure 7.11: The overall workflow for PatchWeave

First, during patch extraction (Lines 1-2 in Algorithm 4), `PatchWeave` takes as input program P_a, program P_b and outputs the difference of the two programs in two formats: textural difference and AST structural difference. A textual difference ($text\_d$) between the two programs provides a list of diff locations in terms of source file paths and line numbers. We use information from the textual difference to identify potentially changed locations which are relevant for the patch, and use this information to generate the AST in a granular level instead of generating AST for the complete program. Since the difference between the two programs may contain modifications that are irrelevant for the bug fix, we use trace based filtering to identify the correct patch from the difference of the two programs. In the initial steps in Algorithm 4, we preprocess the diff locations using the traces generated by executing P_a, P_b and P_c for the failing test case t_F. From the textual difference at each diff location, we identify a *code chunk* which represents the textual difference from P_a to P_b at the given diff location. Using GumTree, we obtain the AST structural difference at each diff location from the two programs P_a and P_b which captures the transformation from P_a to P_b with respect to its abstract syntax tree. The objective of this phase is to correctly identify the patch which fixes the bug, expressed in the form of an AST transformation script. Since the transformation of the AST abstracts concrete identifiers and capture the difference at a fine-grained level, `PatchWeave` could adapt the patch to different contexts.

Second, during patch localization (Lines 3-5 in Algorithm 4), `PatchWeave` computes a patch location for the transplantation of the filtered patch. `PatchWeave` divides the task of patch localization into two sub-tasks, (1) finding the patch function, and (2) finding the patch location within the identified function. At Line 3 in Algorithm 4, the `EstimateDivergentPoint` method uses concolic execution [119] to obtain the partial path conditions of P_a, P_b and P_c for the input t_F to find a divergent point in P_c (i.e. d_c) similar to the divergent point observed in P_a (i.e. d_a) with respect to the filtered patch. Once we identified a similar divergent point in P_c, `PatchWeave` iterates over the trace of the target program P_c to find a patch location. We locate the patch function using `FindPatchFunction` method, which uses the estimated divergent location and the variables used in the code chunk to search for a candidate patch function. First, it filters the functions invoked by P_c in $trace_c$ starting from the estimated divergent point d_c. Then, it finds the first function in the filtered list, which can be mapped to variables used in the code chunk into variables used in the function. Similarly, `FindPatchLoc` method searches for a patch location within the identified patch function using live analysis of the variables mapped by `FindPatchFunction` method. Finally, patch localization provides the identified insertion location for the patch in terms of a target function and the position within the target function to insert the patch, which also gives the context information (i.e. variable mapping) required to translate the patch from the namespace of P_a into the namespace of P_c.

In the patch adaptation phase (Line 6 in Algorithm 4), `PatchWeave` obtains a

Algorithm 4: PatchWeave Algorithm

```
input  : Buggy version of Donor $P_a$
         Fixed version of Donor $P_b$
         Buggy version of Target $P_c$
         Failing test case $t_F$
output: Fixed version of Target $P_d$ or $\phi$
    $trace_a \leftarrow \text{Trace}(P_a, t_F)$
    $trace_b \leftarrow \text{Trace}(P_b, t_F)$
    $trace_c \leftarrow \text{Trace}(P_c, t_F)$
    $text\_d \leftarrow \text{Diff}(P_a, P_b)$
    $text\_d\_filtered \leftarrow \text{FilterDiff}(text\_d, trace_a, trace_b)$
    $text\_d\_filtered \leftarrow text\_d\_filtered.\text{ reverse}()$
    while $text\_d\_filtered$ do
        /* Patch Extraction                                        */
        $d_a, code\_chunk \leftarrow text\_d\_filtered.\text{ pop}()$
        $transformation\_script \leftarrow \text{GumTree}(code\_chunk)$
        /* Patch Localization                                      */
        $d_c \leftarrow \text{EstimateDivergentPoint}(d_a, trace_c)$
        $candidate\_function, var\_map \leftarrow \text{FindPatchFunction}(d_c, trace_c)$
        $candidate\_loc \leftarrow$
          $\text{FindPatchLoc}(candidate\_function, var\_map, transformation\_script)$
        /* Patch Adaptation                                        */
        $translated\_script \leftarrow \text{TranslateScript}$
          $(transformation\_script, d_a, d_c)$
        /* Patch Enforcement                                       */
        $P_d \leftarrow \text{Transform}(translated\_script, var\_map, candidate\_loc)$
    end
    if $SyntaxCheck(P_d)$ then
        return $P_d$
    end
    return $\phi$
```

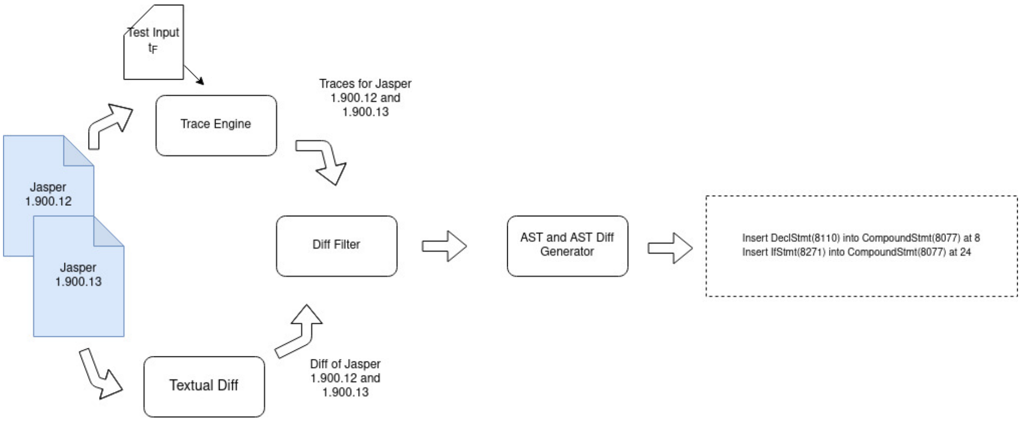
translated patch for P_c. The first step of this phase is to obtain an AST transformation which can convert P_c into P_d. At this point, we have the AST transformation script for P_a, and we have computed the target location for the insertion of the patch. `TranslateScript` method in Line 6 uses an AST node matching algorithm to obtain a mapping between the target function identified from patch localization phase and the AST of P_a. Using this mapping we can translate the AST transformation script into the context of the target program P_c. Second, using the variable map computed earlier in patch localization phase we translate the concrete identifiers (i.e., variable names and data structures) from P_b to P_c.

In patch enforcement (Line 7 in Algorithm 4), `PatchWeave` uses the mapping of concrete identifiers and the adapted AST transformation to weave the patch into the identified patch location in P_c. In this phase, dependency analysis is used to locate and evolve the patch such that all required dependencies (i.e. header files, macro definitions, etc) for the patch are transplanted such that the patch is syntactically correct. Finally, after successful transplantation, we validate our transplanted patch. Given the patched version of P_c, we call it P_d, we validate P_d as follows. First, we use a syntax checker (SyntaxCheck(P_d) in Algorithm 4) which performs static analysis on P_d with a set of syntax-rules to fix any found plausible errors (i.e., unused variables, implicit conversion). Second, we recompile the patched target application and check that the build is successful without any syntactical errors. Third, we execute the patched application on the bug-triggering input to verify that the patch has successfully eliminated the vulnerability for that input. Finally, to check for the deviation of P_d's behavior from P_c's behavior, we perform differential fuzz testing over 100 generated test cases using the input t_F as the seed.

7.4.1 Patch Extraction

`PatchWeave` uses trace based filtering to narrow down the changes from P_a to P_b which only consist of code modifications relevant for the bug. The dynamic profiler used by `PatchWeave` during trace collection is a modified version of KLEE [16]. `PatchWeave` executes the programs P_a and P_b in LLVM IR instructions with the vulnerability triggering input t_F until the buggy location is reached or the program crashes. The modified version of KLEE uses the debugging information in the program to translate each instruction executed to a location in a source file. Using the traces collected, combined with the textual difference obtained from the difference of the two source codes of P_a and P_b, we filter the differences not witnessed in the trace. The underlying assumption is that any modification required for the fix of the bug should be executed in the patched version of the donor P_b. In essence, what we extract as the patch is the necessary and sufficient modification required for the fix.

The filtered patch can be viewed as a code difference composed of multiple code chunks across different locations, each of which is a contiguous sequence of lines corresponding to a sequence of insertions, deletions or both. Once we identified the necessary code chunks required for the patch, we capture the modification as a transformation of an abstract syntax tree. `PatchWeave` constructs an Abstract Syntax Tree (AST) for the function that contains the identified chunk in both P_a and P_b. Using a tree difference algorithm GumTree[34], we generate a transformation script for the ASTs constructed earlier. This transformation script captures the modifications of line insertion, deletion or both in the context of the AST. The set of such transformations at each identified chunk is the output of this phase.


Test Input
t_F
Trace Engine
Traces for Jasper 1.900.12 and 1.900.13
Jasper 1.900.12
Jasper 1.900.13
Diff Filter
AST and AST Diff Generator
Textual Diff
Diff of Jasper 1.900.12 and 1.900.13

```
Insert DeclStmt(8110) into CompoundStmt(8077) at 8
Insert IfStmt(8271) into CompoundStmt(8077) at 24
```

Figure 7.12: Patch extraction phase of PatchWeave

In our motivational example in Section 7.2, for the two versions of our donor program JasPer, we obtained an AST script as depicted in Figure 7.12. Although the original developer patch includes a statement replacement in line 1238 in Figure 7.5a, this statement is never executed for the input t_F in the fixed version of

JasPer. Hence, we filter this statement and only include the insertion of the variable declaration and the if-condition which checks for the overflow.

7.4.2 Patch Localization

For each code chunk collected in the previous phase, the patch localization step aims to search for the location in P_c to insert the respective code chunk. Algorithm 5 explains how `PatchWeave` estimates a similar location in P_c. The diff location of each chunk is a divergent point in P_a since the execution of the t_F in P_a and P_b start to differ at this location. For each such divergent point d_a, we first calculate the partial path condition of the failing input t_F in P_a at d_a. This is the path condition of the trace π_a^F of input t_F in program P_a up to and including the divergent point. We estimate a similar location in P_c using partial path condition dominance, on the trace of input t_F in program P_c. We map a divergent point d_a to the earliest point d_c in the execution trace of t_F in program P_c which satisfies

$$dom(ppc(P_c, t_F, d_c), ppc(P_a, t_F, d_a))$$

where *dom* is a dominance relation defined in Def. 6. Note that the two programs although semantically similar may have different input verification, hence a superset of the input bytes of d_a in P_c may not exist. For instance, in our motivational example, the divergent point is at "libjasper/jpc/jpc_dec.c:1234" and the input bytes appearing in the partial path condition at this points are shown below in "input_bytes_a". Similarly, OpenJPEG 1.5.1 input bytes appearing at the full path condition are given below in "input_bytes_c".

```
input_bytes_a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18,
    19, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 62, 63, 64,
     65, 66, 67, 68, 69, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90,
    93, 94, 95, 96, 97, 98, 99, 100, 109, 110, 111, 112, 113, 114, 115, 116, 117,
    118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129, 130, 131, 132, 133,
     134, 135, 136, 137, 138]

input_bytes_c = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18,
    19, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 58, 62, 63,
     64, 65, 66, 67, 68, 69, 70, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88,
    93, 94, 95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108, 109,
    110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125,
     126, 127, 128, 129, 130, 131, 132, 133, 134, 135, 136, 137, 138]
```

Note that input bytes [89, 90] are missing in “input_bytes_c”, because OpenJPEG 1.5.1 does not use these bytes at any control location. Hence, we need to rule out any byte that does not appear in P_c before checking for the dominance relation. Algorithm 5 explains how PatchWeave overcomes this issue by taking the intersection of input bytes of the full path condition of P_c and the input bytes of the partial path condition at divergent point d_a in P_a. Once the filtered input bytes for the dominance relation is obtained, PatchWeave iterates through all control locations in P_c in a reverse order (i.e. starting from the last execution location) and traverses until the dominance relation does not hold, which would give us the estimated divergent location. We iterate in reverse order to be time efficient because the patch location is much closer to the crash/buggy location.

Once we identify a location in P_c the patch can be inserted at any location between d_c and the crash location, in the execution trace of the failing test in P_c. Note that there can be multiple divergent points due to multiple changes made between P_a and P_b. Once a location d_c is found for a given d_a, the adaptation of the code chunk can be applied at any location visited between d_c and the crashing point (in the execution trace of t_F in P_c). The search for this patch location consists of two steps: (1) identifying patch function, and (2) finding the correct patch line within the patch function. Given t_F, our approach performs concolic execution on the three programs (P_a, P_b and P_c) along the paths taken by t_F.

7.4.2.1 Patch function

Algorithm 6 presents how PatchWeave finds the candidate patch function. The insertion point is bounded between d_c (divergent point) and l_c (crash location), a candidate function is any function invoked between d_c and l_c in the execution trace of t_F in P_c. In Algorithm 6, list_functions denotes all such functions executed between d_c and l_c in the execution trace of t_F in P_c. PatchWeave traverses through each such function and generates symbolic expressions for the variables to find a mapping between the variables in the code chunk at the divergent point d_a. Given a function f_c in P_c, our goal is to map each variable in the code chunk to variables in function f_c in P_c. Thus, for each variable in the code chunk, if a variable can be

Algorithm 5: Estimating a divergent point (estimateDivLoc(d,p))

```
routine: getPartialPathCondition(p) takes a program location and outputs
         the partial path condition
         extractInputBytes(p) takes a partial path condition and outputs
the input bytes in the path condition
         extractControlLocations(p) takes a program trace and outputs
control locations
input  : a divergent point in $P_a$ ($d_a$), execution trace of $P_c$ for input
         $t_F$ ($\pi_c^F$)
output : a location in $P_c$ or $\phi$

    $ppc_a \leftarrow getPartialPathCondition(d_a)$
    $bytes\_list_a \leftarrow extractInputBytes(ppc_a)$
    $list_{control} \leftarrow extractControlLocations(\pi_c^F)$
    $location_{end} \leftarrow list_{control}.last()$
    $ppc_{end} \leftarrow getPartialPathCondition(location_{end})$
    $bytes\_list_{end} \leftarrow extractInputBytes(ppc_{end})$
    $bytes\_list_a \leftarrow bytes\_list_a \cap bytes\_list_{end}$
    /* Iterate through the control locations in $P_c$ to find a
       location which satisfies the dominance relation        */
    $estimate\_loc \leftarrow \phi$
    while $list_{control}$ do
        $location_c \leftarrow list_{control}.pop()$
        $ppc_c \leftarrow getPartialPathCondition(location_c)$
        $bytes\_list_c \leftarrow extractInputBytes(ppc_c)$
        if $bytes\_list_a \sqsubseteq bytes\_list_c$ then
            $estimate\_loc \leftarrow location_c$
        else
            return $estimate\_loc$
        end
    end
    return $estimate\_loc$
```

mapped to a variable in f_c of P_c with the same symbolic expression[4](getMap(l,f) in Algorithm 6), we consider f_c as a candidate patch function. In our motivational example, function j2k_read_siz is our candidate insertion function since it is within the range of the estimated divergent point and the crashing location, and it includes variables that match the symbolic expressions of the variables in the code chunk that we want to transplant.

Algorithm 6: Finding patch function (FindPatchFunction($d_c, trace_c$))

```
routine: extractCode(p) takes a program location and outputs the program
         statements at given location
         estimateDivLoc(d,p) is the routine described in Algorithm 2
         extractIdentifiers(c) takes a code chunk and output the list of
variable names
         extractFunctions(d,p) takes a program location d, a trace p,
output the functions executed up to d
         getMap(l,f) takes a list of variable names and a function, outputs if
variable mapping is possible
input  : a divergent point in $P_a$ ($d_a$), execution trace of $P_c$ for input
         $t_F$ ($\pi_c^F$)
output : a candidate function $f_c$ in $P_c$ or $\phi$

    $code\_chunk$ ←extractCode($d_a$)
    $d_c$ ←estimateDivLoc($d_a, \pi_c^F$)
    $list\_identifiers$ ← extractIdentifiers($code\_chunk$)
    $list\_functions$ ←extractFunctions($d_c, \pi_c^F$)
    for $function\ f_c\ in\ list\_functions$ do
        if $getMap(list\_identifers, f_c)$ then
            return $f_c$
        end
    end
    return $null$
```

7.4.2.2 Patch Location

Computing the patch location has two variants based on the type of transformation of the patch. If the original patch is modifying an existing statement in P_a, then the objective is to find a similar statement in P_c. However, if the patch is introducing

[4] These symbolic expressions are calculated by the concolic execution of t_F in programs P_b and P_c.

new statements, the patch line will be determined by the liveness property of the variables required for the patch, i.e., the patch line should be a line at which all variables required for the patch holds a value. We identify the line where the transformation of the patch needs to be inserted based on the mapping of the variables in the patch in P_b and to variables in the patch function in P_c. Let the variables appearing in patch be Vars_{ab} and let them be mapped to variables Vars_c in P_c in the previous step of identifying the patch function. Recall that the mapping of Vars_{ab} to Vars_c was achieved by (1) concolic execution of input t_F in P_b and P_c, and then (2) mapping variables based on which variables in P_c have the same symbolic expressions as the symbolic expressions of variables Vars_{ab} in program P_b. Given a patch function f_c in P_c, we filter out all control locations in f_c where the variables Vars_c are not live. Moreover, if the patch function is on the stacktrace when the crash occurs in P_c when executing t_F, we can further narrow down the patch locations to all locations in f_c executed. For any selected candidate patch location in function f_c, we check if Vars_{ab} in the patch possess the same symbolic expressions as the variables Vars_c at the patch location.

7.4.3 Patch Adaptation

The patch adaptation phase processes the translation of the AST transformation script obtained from the patch extraction phase and translates the concrete identifiers to the namespace at the insertion location identified from the patch localization phase. `PatchWeave` first translates the AST transformation script into the context at the insertion location. For each code chunk identified for transplantation at the extraction phase, `PatchWeave` obtains the corresponding AST transformation. Translating the transformation script to the context of P_c involves three steps: node translation, position translation and namespace translation. First, we present the structure of a transformation step in the transformation script as follows:

- **Delete *NodeA***: Delete node *NodeA* from AST_a
- **Insert *NewNode* into *NodeB* at *k***: Inserts the node *NewNode* as the $k-th$ child of node *NodeB* in AST_b

- **Move *NodeA* into *NodeB* at *k***: Moves the node *NodeA* in AST_a to be the k-th child of node *NodeB* in AST_b
- **Update *NodeA* to *NodeB***: Replace the label in node *NodeA* with the label of node *NodeB*
- **Update and Move *NodeA* into *NodeB* at *k***: First update the label of node *NodeA* from the matching node and then move node *NodeA* to the k-th position of node *NodeB*

For instance, in our motivational example in Figure 7.12, one of the transformation action is "Insert IfStmt(8271) into CompoundStmt(8077) at 24", which describes as inserting the node of type "IfStmt" identified by the id 8271 in P_b, into the node of type "CompoundStmt" identified by the id 8077 in P_b.

7.4.3.1 Node Translation

Given a transformation of an AST node in P_a into a AST node in P_b, we want to replace the node with a node in P_c and apply the same transformation. For this purpose, we use the tree differencing algorithm GumTree[34] implementation on LLVM AST. GumTree maps nodes in the two input ASTs based on certain heuristics [34]. It outputs a set of mapped nodes denoted as $(X, Y) = (X_1, Y_1), ..., (X_i, Y_i)$ where $X = X_1, .., X_i$ and $Y = Y_1, .., Y_i$ are the mapped nodes in the two ASTs and 'i' is the number of mapped nodes. We use this technique to generate a mapping between P_a and P_c. Specifically, we generate the AST of the function which contains the AST node of P_a, and the AST of the candidate function in P_c identified in the previous phase. Once we obtain the mapping of nodes in the ASTs of P_a and P_c, `PatchWeave` uses this mapping to translate the transformation script obtained in the extraction phase.

Figure 7.13 shows the AST nodes translation using our motivational example. The arrows indicate a mapping identified by GumTree from one AST to another. To translate the transformation operation into P_c, we first translate the operation into the context of P_a and then translate back from P_a into P_c. In Figure 7.13, node 6 and 8 in the AST of JasPer 1.900.13 is mapped to the nodes 6 and 7 in the AST of JasPer 1.900.12 respectively. Similarly, node 6 and 7 in the AST of JasPer 1.900.12 is mapped to the nodes 5 and 6 in AST of OpenJPEG 1.5.1. Thus, giving

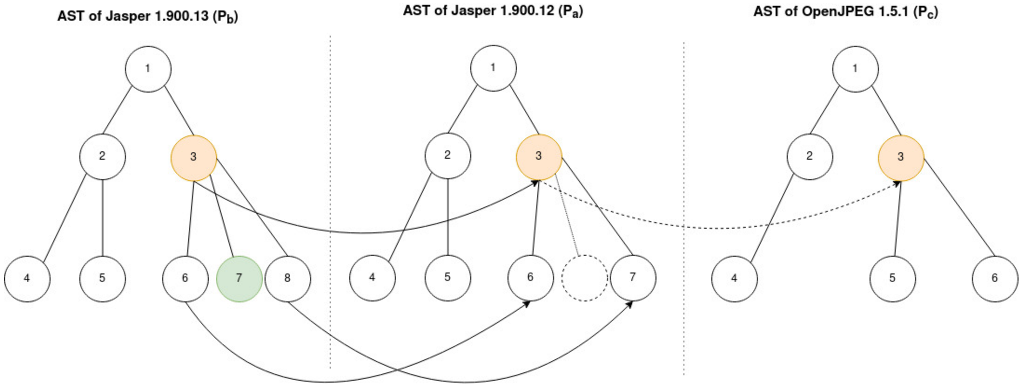

AST of Jasper 1.900.13 (P_b)
AST of Jasper 1.900.12 (P_a)
AST of OpenJPEG 1.5.1 (P_c)
1
2
3
4
5
6
7
8
1
2
3
4
5
6
7
1
2
3
4
5
6

Figure 7.13: AST Node Mapping in Patch Adaptation Phase

us the translation of node 6 and 8 from AST of JasPer 1.900.13 into node 5 and 6 in OpenJPEG 1.5.1 respectively.

Let us consider in our motivational example where we transplant an "if" statement from P_b into P_c. Figure 7.13 shows the translation of the AST nodes. The diagram depicts the node translation for the transformation operation of "Insert IfStmt(8271) into CompoundStmt(8077) at 24" discussed earlier. The green node in Figure 7.13 represents the "IfStmt" node which need to be inserted while the yellow nodes represent the "CompoundStmt" nodes in all three programs. First, we map the "CompoundStmt(8077)" node from P_b into the "CompoundStmt(8077)" node in P_a, and then "CompoundStmt(8077)" node from P_a into the "CompoundStmt(3886)" node in P_c as shown in Figure 7.13. At this point of the adaptation, the transformation operation is "Insert IfStmt(8271) into CompoundStmt(3886) at X". Next, we adjust the position of the transformation related to the translated AST node "CompoundStmt(3886)" in P_c.

7.4.3.2 Position Translation

Given a translation of nodes for the transformation action, we convert the position relative to the translated AST node. For example, some transformations such as INSERT, MOVE requires a position attribute which describes a position relative to the target node in which the transformation would take place. `PatchWeave` uses the patch location identified in previous phase to identify the target AST position by translating the patch location into a relative AST node position.

7.4.3.3 Namespace Translation

To translate the namespace of patch from P_b into P_c, the first step is to generate a mapping between the variables in the patch and P_c. For each variable in the patch, PatchWeave finds the corresponding variable in P_c despite the difference in data structures. As a single variable could have multiple instances due to different invocations, PatchWeave keeps track of all such instances to identify any instance of a variable in P_c that matches with a variable in the patch. For each such instance, PatchWeave records the bit value of KLEE [16], the symbolic expression and the variable name. PatchWeave is relying on the data-type agnostic representation of KLEE to translate values across different data-structures. (i.e. same numeric input represented in a signed integer value and unsigned integer differ). For each variable v in the patch, PatchWeave lists variables u from P_c where any instance of the variable u is equal to the bit value of the given variable v. PatchWeave then filters the variables using equivalence of symbolic expressions of v and u, which guarantees that the two variables are not only equal in terms of the value computed but also in terms of symbolic expressions. For any given variable v in the patch, if there are several variables in P_c having the same bit value and symbolic expression as v, PatchWeave selects the variable whose name has the minimum Levenshtein distance with the name of variable v (e.g., if $array[i]$ is matched against $temp$ and $arr[j]$, PatchWeave selects $arr[j]$ due to the minimum Levenshtein distance).

7.4.4 Patch Enforcement

Once the translation of the variables in the patch into the namespace of P_c is complete, we obtain the translated patch. Although the patch has been translated successfully, we still need to identify and transplant any missing definitions used in the patch, such as a function specific for P_a or any macro definition only defined in P_a. The final step of the transplantation is to analyze each AST node in P_c to identify such missing definitions and ensure that the relevant missing functions can be called from within P_c at the insertion point. For example, if the patch is using a function defined in an external library which is not used in P_c, PatchWeave would include the relevant header files automatically so that the library can be called from P_c. In the case where a custom function defined in P_a is used in the patch, PatchWeave translates the function to match the namespace of P_c as shown in our motivational example.

7.5 Implementation

We have implemented PatchWeave in Python 2.7, in combination with Clang 7.1 [140] and KLEE 1.4 [16]. Clang is used for compiling the AST's and obtaining the LLVM IR for symbolic execution in KLEE. We extend KLEE to support concolic execution for C/C++ programs. We leverage Z3 [142] SMT solver for equivalence checking and LibTooling to enforce the textual edits and for source code instrumentation required in symbolic analysis.

We use the clang AST because it offers source-to-source transformation for C/C++ code and the C++ standard. For example, parenthesis expressions and compile time constants are available in an unreduced form in the AST. This makes Clang's AST a good fit for refactoring tools such as ours. Since we use a compiled AST contrast to a static AST built from the source code, the AST is lightweight but also does not include pragmas which is commonly used in software to produce different code for different environments. For our analysis, we do not require such level of details since we only repair the bug for the vulnerability observed in one environment.

PatchWeave is implemented with 6394 lines of Python code and 4164 lines of C/C++ code not including the modified KLEE used for symbolic analysis.

7.6 Evaluation

First, we evaluate the efficacy of PatchWeave on eight real world applications to transplant fixes for vulnerabilities reported in the CVE database and recurring bugs [106] discovered in our experiments, where each subject is a tuple of (*donor program*, *target program*, *detected error*). In terms of the problem formulation shown in Figure 7.1, donor program corresponds to P_b, target program corresponds to P_c, and detected error corresponds to the failing input t_F. Thus, a patch from P_b is transplanted into P_c. Second, we evaluate the quality of the transplanted patch using differential fuzzing and manually examine the patches for correctness and compare them to the developer fix (if any exist). Third, we compare PatchWeave against state of the art APR tools including F1X [89] and Prophet [79]. Next, we provide our comparison effort with feature transplantation tools and the results. Last, we compare against a syntactic approach for transplantation, namely LASE [93]. Our evaluation aims to address the following research questions:

RQ1 How effective is PatchWeave on real-world programs?

RQ2 Can we localize the correct function for transplantation using our patch localization algorithm?

RQ3 What is the quality of the transplanted code?

RQ4 How effective is our approach compared to Automated Program Repair (APR) tools?

RQ5 How effective is our approach compared to existing transplantation techniques?

RQ6 How effective is a semantic transplantation technique compared with syntactic transplantation techniques?

7.6.1 Experimental Setup

We evaluate PatchWeave on five classes of errors, including integer overflow, division by zero, null pointer dereference, buffer overflow and memory errors. We obtain our subjects from a public repository [2] that contains exploits and steps to reproduce vulnerabilities published in the CVE database. To evaluate the patch transplantation ability of PatchWeave, we studied the transplantation of patches on one application program processing an input format to another application program exercising similar functionality on the same input format. We select subjects based on two criteria: (1) they have exploits reported in the referenced public repository [2], and (2) they should be popular C and C++ open-source programs which have been cited in literature [1, 67]. Table 7.3 shows the summary of each selected subject, together with a short description about the functionality of the subject, lines of code (LOC), the range of the program versions and the number of versions we evaluated.

Table 7.3: Experiment subjects used in PatchWeave evaluation

Name	Description	LOC	Version Range	Count
JasPer	Image manipulations	26k	1.900.2 - 2.0.14	44
OpenJPEG	JPEG 2000 image manipulations	200k	1.4 - 2.1.1	8
LibWebP	WebP image manipulations	67k	0.1.2 - 1.0.2	14
LibTIFF	TIFF images processing	70k	4.0.0 - 4.0.9	10
LibMing	SWF processing	66k	0.4.3 - 0.4.8	6
Libsndfile	Audio manipulation	52k	1.0.25 - 1.0.28	4
Libzip	ZIP archive processor	13k	1.0.0 - 1.5.2	14
WavPack	Lossless Wave file compressor	33k	4.40.0 - 5.1.0	12

For our experiments, we select vulnerabilities and their corresponding exploits from reported issues in the projects listed in Table 7.3 using the following criteria: (1) the vulnerability should be exploitable and its report includes a proof of exploit, (2) the vulnerability should be fixed and verified by the developer, and (3) the vulnerability should be reported in the CVE database between 2016-2018. For each vulnerability collected, we run against each of the subjects (each version of each program listed in Table 7.3) to find any similar vulnerability exhibiting in a different program other than the one it was reported on. Any two pairs that exhibit similar vulnerability (Def 1) are considered for evaluation under two criteria. 1) if the two programs are not from the same project, we select the latest version of the target project in the range we presented in Table 7.3 or 2) if both programs are from the same project, it implies a backporting pair. For backporting, we select the oldest version in the range we presented in Table 7.3 which exhibits the same vulnerability. We further filtered commits which did not compile as we cannot verify if the vulnerability has been fixed for the exploit. Note that the exploits we collected can trigger vulnerabilities that are common to two different programs which are not relevant for the original program it was reported with. For example, an exploit which was reported with OpenJPEG, could trigger a vulnerability which is common to Jasper and Libtiff.

All experiments are conducted on a Dell PowerEdgeR530 with Intel(R) Xeon(R) CPU E5-2660 processor and 64GB RAM. We use Docker [141] containers to exploit and repair the vulnerable applications.

7.6.2 Experimental Results

7.6.2.1 Efficacy of PatchWeave

We select a set of real-world applications and CVE bug reports as described in Section 7.6.1 and run `PatchWeave` to transplant patches from the donor program to the target program, in the identified pair list. We validate the efficacy of `PatchWeave` by comparing our results to developer patches (if any exist) for the bugs as the ground truth. We extract the developer patches from the bug reports or the commits provided by the developers. Table 7.4 presents the results of our experiments. The "Bug ID" column specifies an identifier of the bug if the bug has been reported. If the bug has been reported in CVE we indicate the CVE identifier or if the bug has been reported in the project bug tracker, we indicate the bug identifier. Some bugs have been fixed without being reported hence may not have a bug ID, which is indicated by 'N/A'. The "Exploit ID" column indicates the CVE where we obtained the exploits / test cases (t_F). The "Donor" column specifies the program name and version of the donor, while the "Target" column shows the same information for the target program. The "Target Location" column indicates the source code location in the target program that contains the vulnerability. The "Error" column specifies the type of vulnerability. The "Patch Commit" column represents the commit id for the patch in the donor program and "Patch Class" shows the classes of patches defined in Section 7.3. The "Time (min)" column shows the total time taken in minutes for PatchWeave to fix the error/vulnerability by transferring the patch, starting from patch extraction to patch verification or ✗ to indicate the patch transplantation was unsuccessful.

The "Diff. Fuzz." column denotes the results of differential fuzz testing in the form x/y, where x is the number of test cases where P_c results in a crash and P_d gracefully exits; similarly y is the number of test case where P_d crashes or produce a different output than P_c. The "Patch Size" column denotes the modified lines of code in the transferred patch (e.g., if an expression within an if-statement is modified, we consider "Patch Size"=1). The "Function Hops" column is a measure of efficiency in finding the patch function, (e.g., "Function Hops" = 1 means that we find the patch function at the first iteration of the loop in Algorithm 6). The "Patch Similarity" column denotes the patch quality of PatchWeave relative to

Table 7.4: Summary of PatchWeave experiment results

ID	Bug ID	Exploit ID	Donor	Target	Target Location	Error	Patch Commit	Patch Class	Time (min)	Diff. Fuzz.	Patch Size	Function Hops	Patch Similarity
1	Bug-169	CVE-2016-8691	JasPer-1.900.3	OpenJPEG-1.5.1	int.h@87	DZ	d8c2604	III	5.0	37/0	3	1	C2
2	CVE-2016-8691	CVE-2016-8691	OpenJPEG-1.5.2	JasPer-1.900.2	*jpc_dec.c*@1194	DZ	e55d5e2	III	8.0	22/0	3	4	C2
3	N/A	CVE-2016-9387	JasPer-1.900.13	OpenJPEG-1.5.1	j2k.c@560	IO	d91198a	IV	7.5	5/0	27	1	C2
4	CVE-2016-9387	CVE-2016-9387	OpenJPEG-1.5.2	JasPer-1.900.12	*jpc_dec.c*@1234	IO	6e0162a	III	8.5	42/0	3	1	C2
5	Bug-155	CVE-2017-6850	JasPer-2.0.12	OpenJPEG-1.5.1	cio.c@146	NPD	7692d6d	III	✗	-/-	-	-	-
6	CVE-2016-6850	CVE-2017-6850	OpenJPEG-1.5.2	JasPer-1.900.30	*jas_malloc.c*@111	NPD	7720188	III	15.0	9/0	4	1	C2
7	N/A	CVE-2016-8692	JasPer-1.900.3	OpenJPEG-1.3	int.h@87	DZ	3c55b39	III	3.5	52/0	3	1	C2
8	CVE-2016-8692	CVE-2016-8692	OpenJPEG-1.4	JasPer-1.900.2	*jpc_dec.c*@1196	DZ	f4d394d	III	5.5	20/0	3	4	C2
9	N/A	CVE-2016-9387	JasPer-1.900.14	OpenJPEG-2.1.0	j2k.c@2099	UIO	ba2b9d00	III	6.0	42/0	4	1	C2
10	N/A	CVE-2016-9387	OpenJPEG-2.1.1	JasPer-1.900.13	*jpc_dec.c*@1244	UIO	58fc8645	III	18.0	46/0	13	2	C2
11	Bug-312	CVE-2016-9262	OpenJPEG-1.5.1	LibWebP@0.3.0	jpegdec.c@251	SIO	6280b5ad	III	2.5	90/0	4	1	C2
12	N/A	CVE-2016-9830	JasPer-1.900.4	LibWebP@0.2.0	cwebp.c@120	ShO	6109f6a	II	5.0	99/0	2	1	C1
13	N/A	CVE-2016-9830	LibWebP@0.3.0	JasPer-1.900.3	*mif_cod.c*@394	ShO	7a650c6a	I	11.0	98/0	1	13	C1
14	CVE-2016-9390	CVE-2016-9390	OpenJPEG-1.5.2	JasPer-1.900.13	$jpc_m ct.c$@151	HBO	69cd4f9	III	✗	-/-	-	-	-
15	Bug-297	CVE-2016-9390	JasPer-1.900.14	OpenJPEG-2.1.0	libopenjpeg/mct.c@84	HBO	dee11ec	IV	✗	-/-	-	-	-
16	CVE-2016-9393	CVE-2016-9393	JasPer-1.900.17	OpenJPEG-1.5.2	j2k.c@447	UIO	f7038068	III	12.0	42/0	3	1	N.A.
17	CVE-2016-8884	CVE-2016-8884	LibTiff-3.8.0	Jasper-1.900.8	*bmp_dec.c*@394	MWE	50373d7d	III	26.0	18/0	8	23	C2
18	N/A	CVE-2017-6828	Libsndfile-1.0.26	WavPack-5.1.0	common.c@992	ShO	3e91aaf7	III	✗	-/-	-	-	-
19	CVE-2016-9387	CVE-2016-9387	Jasper-1.900.13	Jasper-1.900.2	*jpc_dec.c*@1206	IO	d91198a	II	8.0	34/0	29	1	C1
20	CVE-2016-9265	CVE-2016-9265	LibMing-0.4.8	LibMing-0.4.6	listmp3.c@187	DZ	b0704f80	I	5.5	98/0	2	1	C1
21	CVE-2016-9266	CVE-2016-9266	LibMing-0.4.8	LibMing-0.4.6	listmp3.c@94	NS	2e5a98a0	I	5.5	29/0	17	1	C1
22	Bugzilla-2634	CVE-2017-14039	LibTiff-4.0.8	LibTiff-4.0.0	tiff2ps.c@2443	HBO	5ed9fea5	I	26.0	75/0	5	18	C1
23	CVE-2017-8365	CVE-2017-8365	Libsndfile-master	Libsndfile-1.0.26	src/pcm.c@670	GBO	fd0484ab	I	16.0	73/0	9	3	C1
24	CVE-2017-14107	CVE-2017-14107	Libzip-1.3.0	Libzip-1.1.2	*zip_dirent.c*@108	MAF	9b46957e	I	42.0	90/0	4	13	C1

Division By Zero Error {DZ - Divide by Zero}, *Integer Overflow* {IO - Integer Overflow, UIO - Unsigned Integer Overflow, SIO - Signed Integer Overflow}, *Memory Error* {NPD - Null Pointer Dereference, MAF - Memory Allocation Failure, MWE - Memory Write Error}, *Shift Overflow* {ShO - Shift Overflow, NS - Negative Shift}, *Buffer Overflow* {HBO - Heap Buffer Overflow, GBO - Global Buffer Overflow}

human-written patches (defined in Section 7.6.2.3).

Overall, PatchWeave has successfully fixed the errors for all evaluated pairs via patch transplantation, except for four errors. PatchWeave fails to fix ID 5 in Table 7.4 because the patch for JasPer includes program-specific changes that cannot be translated to OpenJPEG as intended. Specifically, the changes involve refactoring of existing functions and modifications to internal function calls, which are irrelevant for the bug-fix. Similarly, the developer patch in ID 18 contains changes for multiple bug fixes so our approach fails to extract the specific patch for the bug. In ID 14 and 15 of Table 7.4, the transplantation was unsuccessful due to the higher number of iterations executed within the execution of t_F which produced a large symbolic path condition in which checking of partial path conditions became computationally infeasible.

Artifact and Tool Release Experiment results in the form of generated patches and developer patches are publicly available at our website[5]. Other experimental data is publicly available at Docker Hub via docker image rshariffdeen/patchweave:experiments. Meanwhile, our tool is publicly available on Github[6].

7.6.2.2 Effectiveness of patch localization

We evaluate the effectiveness of the patch localization technique in `PatchWeave` (i.e. partial path condition dominance relations defined in Definition 6). Table 7.5 summarizes the efficacy of `PatchWeave` for patch localization. The "Donor" column specifies the program name and version of the donor, while the "Target" column shows the same information for the target program. The "Error" column specifies the type of vulnerability and "Patch Class" shows the classes of patches defined in Section 7.3. The "Time (min)" column shows the total time taken in minutes for `PatchWeave` to fix the error/vulnerability by transferring the patch, starting from patch extraction to patch verification or ✗ to indicate the patch transplantation was unsuccessful.

"In Top-3?" column and "In Top-5?" column indicate if the patch function has been located after localization within 3 hops and 5 hops respectively. The "Localized

[5] https://patchweave.github.io/

[6] https://github.com/rshariffdeen/PatchWeave

Function Hops" column indicates the absolute measure of efficiency in finding the patch function (same as "Function Hops" column in Table 7.4). Similarly, "Non-Localized Function Hops" column indicates the number of hops required to iterate in order to find the identified patch function without patch localization. "Filter Count" represents the number of functions filtered from the search space using patch localization and "Reduction Ratio" represents the efficiency of the patch localization in terms of the number of hops saved as a percentage of the total number of hops without patch localization. The percentage is calculated in the form x/y, where "x" is the number of functions filtered and "y" is the total number of function hops required when not using patch localization.

`PatchWeave` successfully locates the patch function for 20 test-cases presented in Table 7.5 among these 11 instances are the first candidate function, and 13 hits the Top-3 selection and 15 hits the Top-5 selection.

`PatchWeave` can correctly identify the insertion points for all patches transplanted for all evaluated (*donor, target*) pairs. In general, there could be several potential insertion points for a given patch, our algorithm has successfully identified one of these insertion points. On average, our approach requires six iterations to find the insertion points for all evaluated patches ("Function Hops" column in Table 7.4 and "Localized Function Hops" column in Table 7.5). As our approach can automatically identify all insertion points within a reasonable number of iterations, this serves as evidence for the effectiveness of our localization algorithm using the partial path condition dominance relationship.

7.6.2.3 Quality and diversity of patches

Given a developer-written patch $Patch_{dev}$, an automatically transplanted patch $Patch_{auto}$, we measure patch quality using the criteria:

(C1) Syntactically Equivalent. $Patch_{auto}$ is "Syntactically Equivalent" if $Patch_{auto}$ and $Patch_{dev}$ are syntactically the same.

(C2) Semantically Equivalent. $Patch_{auto}$ is "Semantically Equivalent" if $Patch_{auto}$ and $Patch_{dev}$ are not syntactically the same but could be refactored to produce the same semantic behavior.

Table 7.5: Effectiveness of patch localization in PatchWeave

ID	Donor	Target	Error	Patch Class	Time (min)	In Top-3?	In Top-5?	Localized Function Hops	Non-Localized Function Hops	Filter Count	Reduction Ratio
1	JasPer-1.900.3	OpenJPEG-1.5.1	DZ	III	5.0	✓	✓	1	18	17	17/17 = 100%
2	OpenJPEG-1.5.2	JasPer-1.900.2	DZ	III	8.0	✗	✓	4	65	61	61/64 = 95%
3	JasPer-1.900.13	OpenJPEG-1.5.1	IO	IV	7.5	✓	✓	1	34	33	33/33 = 100%
4	OpenJPEG-1.5.2	JasPer-1.900.12	IO	III	8.5	✓	✓	1	82	81	81/81 = 100%
5	JasPer-2.0.12	OpenJPEG-1.5.1	NPD	III	✗	-	-	-	-	-	-
6	OpenJPEG-1.5.2	JasPer-1.900.30	NPD	III	15.0	✓	✓	1	33	32	32/32 = 100%
7	JasPer-1.900.3	OpenJPEG-1.3	DZ	III	3.5	✓	✓	1	18	17	17/17 = 100%
8	OpenJPEG-1.4	JasPer-1.900.2	DZ	III	5.5	✗	✓	4	65	61	61/64 = 95%
9	JasPer-1.900.14	OpenJPEG-2.1.0	UIO	III	6.0	✓	✓	1	57	56	56/56 = 100%
10	OpenJPEG-2.1.1	JasPer-1.900.13	UIO	III	18.0	✓	✓	2	70	68	68/69 = 98%
11	OpenJPEG-1.5.1	LibWebP@0.3.0	SIO	III	2.5	✓	✓	1	82	81	81/81 = 100%
12	JasPer-1.900.4	LibWebP@0.2.0	ShO	II	5.0	✓	✓	1	7	6	6/6 = 100%
13	LibWebP@0.3.0	JasPer-1.900.3	ShO	I	11.0	✗	✗	13	13	0	0/12 = 0%
14	OpenJPEG-1.5.2	JasPer-1.900.13	HBO	III	✗	-	-	-	-	-	-
15	JasPer-1.900.14	OpenJPEG-2.1.0	HBO	IV	✗	-	-	-	-	-	-
16	JasPer-1.900.17	OpenJPEG-1.5.2	UIO	III	12.0	✗	✗	1	34	33	33/33 = 100%
17	LibTiff-3.8.0	Jasper-1.900.8	MWE	III	26.0	✗	✗	23	46	23	23/45 = 51%
18	Libsndfile-1.0.26	WavPack-5.1.0	ShO	III	✗	-	-	-	-	-	-
19	Jasper-1.900.13	Jasper-1.900.2	IO	II	8.0	✓	✓	1	79	78	78/78 = 100%
20	LibMing-0.4.8	LibMing-0.4.6	DZ	I	5.5	✓	✓	1	2	1	1/1 = 100%
21	LibMing-0.4.8	LibMing-0.4.6	NS	I	5.5	✓	✓	1	2	1	1/1 = 100%
22	LibTiff-4.0.8	LibTiff-4.0.0	HBO	I	26.0	✗	✗	18	76	58	58/75 = 77%
23	Libsndfile-master	Libsndfile-1.0.26	GBO	I	16.0	✓	✓	3	17	14	14/16=87.5%
24	Libzip-1.3.0	Libzip-1.1.2	MAF	I	42.0	✗	✗	13	25	12	12/24 = 50%

Division By Zero Error {DZ - Divide by Zero}, *Integer Overflow* {IO - Integer Overflow, UIO - Unsigned Integer Overflow, SIO - Signed Integer Overflow}, *Memory Error* {NPD - Null Pointer Dereference, MAF - Memory Allocation Failure, MWE - Memory Write Error}, *Shift Overflow* {ShO - Shift Overflow, NS - Negative Shift}, *Buffer Overflow* {HBO - Heap Buffer Overflow, GBO - Global Buffer Overflow}

Table 7.4 shows that our approach could successfully generate patches of comparable quality to the developer-written patches (i.e., eight generated patches are "Syntactically Equivalent" and 11 are "Semantically Equivalent"). We attribute the success of PatchWeave in generating patches that are of comparable quality to developer-written patches to the code reuse advocated by our approach in the form of patches. The "Patch Size" column in Table 7.4 indicates that our approach is effective in transplanting compact patches (i.e., they are all expressible within 1–29 lines of code). Meanwhile, the "Diff. Fuzz." column shows that all of the transplanted patches have been validated through differential fuzzing.

In terms of the patch types covered, the "Patch Type" column shows that `PatchWeave` could successfully transplant patches for all types of patches defined in Table 7.1. In terms of the class of errors covered, the "Error" column shows that `PatchWeave` could successfully transplant patches for all evaluated five classes of vulnerabilities, namely: buffer overflow, integer overflow, divide-by-zero, memory errors (including null pointer dereferences) and shift overflows.

7.6.2.4 Comparison with APR

Although not directly comparable, automated program repair can be used to directly repair program bugs instead of transplanting from a different program. Hence, we compare our technique with two state-of-the-art program repair techniques to show how transplantation addresses the limitations of program repair (i.e. bounded search space, overfitting problem). Comparison results are shown in Table 7.6. The "Time" column shows the total time taken in minutes for each tool to fix the error/vulnerability, or ✗ to indicate the repair was unsuccessful. The "Fuzz" column denotes the results of differential fuzz testing in the form x/y, where x is the number of test cases where P_c results in a crash and P_d gracefully exits; similarly y is the number of test case where P_d crashes or produce a different output than P_c.

F1X [89]: We evaluate our benchmark with $F1X$, one of the latest semantic-based automated program repair tool for C programs; the authors of [89] provided us access to the tool at our email request. We choose to evaluate on $F1X$ because it represents a state-of-the-art program repair tool which has been shown in prior work to be more efficient and effective than several search based and semantic program repair tools. $F1X$ uses test-equivalence relations to partition patch candidates,

Table 7.6: PatchWeave comparison with program repair techniques

ID	Donor	Target	Error	Patch Class			F1X		Prophet	
					Time	Fuzz	Time	Fuzz	Time	Fuzz
1	JasPer-1.900.3	OpenJPEG-1.5.1	DZ	Class-III	5.0	37/0	0.16	20/13	10	69/2
2	OpenJPEG-1.5.2	JasPer-1.900.2	DZ	Class-III	8.0	22/0	1.5	24/0	3.5	23/0
3	JasPer-1.900.13	OpenJPEG-1.5.1	IO	Class-IV	7.5	5/0	0.33	52/0	2	38/7
4	OpenJPEG-1.5.2	JasPer-1.900.12	IO	Class-III	8.5	42/0	6	14/27	2	0/41
5	JasPer-2.0.12	OpenJPEG-1.5.1	NPD	Class-III	✗	-/-	0.16	11/3	✗	-/-
6	OpenJPEG-1.5.2	JasPer-1.900.30	NPD	Class-III	15.0	9/0	0.5	8/0	3	0/9
7	JasPer-1.900.3	OpenJPEG-1.3	DZ	Class-III	3.5	52/0	0.25	30/29	1	1/69
8	OpenJPEG-1.4	JasPer-1.900.2	DZ	Class-III	5.5	20/0	1.5	33/0	3.5	20/0
9	JasPer-1.900.14	OpenJPEG-2.1.0	UIO	Class-III	6.0	42/0	0.67	37/49	✗	-/-
10	OpenJPEG-2.1.1	JasPer-1.900.13	UIO	Class-III	18.0	46/0	6	35/7	✗	-/-
11	OpenJPEG-1.5.1	LibWebP@0.3.0	SIO	Class-III	2.5	90/0	✗	-/-	3	41/0
12	JasPer-1.900.4	LibWebP@0.2.0	ShO	Class-II	5.0	99/0	0.25	38/0	✗	-/-
13	LibWebP@0.3.0	JasPer-1.900.3	ShO	Class-I	11.0	98/0	0.5	93/4	✗	-/-
14	OpenJPEG-1.5.2	JasPer-1.900.13	HBO	Class-III	✗	-/-	✗	-/-	✗	-/-
15	JasPer-1.900.14	OpenJPEG-2.1.0	HBO	Class-IV	✗	-/-	✗	-/-	✗	-/-
16	JasPer-1.900.17	OpenJPEG-1.5.2	UIO	Class-III	12.0	42/0	1	44/9	1	0/69
17	LibTiff-3.8.0	Jasper-1.900.8	MWE	Class-III	26.0	18/0	0.33	15/0	✗	-/-
18	Libsndfile-1.0.26	WavPack-5.1.0	ShO	Class-III	✗	-/-	✗	-/-	✗	-/-
19	Jasper-1.900.13	Jasper-1.900.2	IO	Class-II	8.0	34/0	7.5	13/26	2	0/42
20	LibMing-0.4.8	LibMing-0.4.6	DZ	Class-I	5.5	98/0	0.33	100/0	2	0 /42
21	LibMing-0.4.8	LibMing-0.4.6	NS	Class-I	5.5	29/0	0.33	100/0	6	94/6
22	LibTiff-4.0.8	LibTiff-4.0.0	HBO	Class-I	26.0	75/0	0.33	0/100	✗	-/-
23	Libsndfile-master	Libsndfile-1.0.26	GBO	Class-I	16.0	73/0	1	66/2	✗	-/-
24	Libzip-1.3.0	Libzip-1.1.2	MAF	Class-I	42.0	90/0	✗	-/-	✗	-/-
	Total			24	20	20	19	8	12	3

Division By Zero Error {DZ - Divide by Zero}, *Integer Overflow* {IO - Integer Overflow, UIO - Unsigned Integer Overflow, SIO - Signed Integer Overflow}, *Memory Error* {NPD - Null Pointer Dereference, MAF - Memory Allocation Failure, MWE - Memory Write Error}, *Shift Overflow* {ShO - Shift Overflow, NS - Negative Shift}, *Buffer Overflow* {HBO - Heap Buffer Overflow, GBO - Global Buffer Overflow}

which leads to significant improvement of the patch generation time without sacrificing patch quality. However, $F1X$, like most existing repair tools, may suffer from the overfitting problem, where a generated patch may be plausible (passing all given tests), but overfitting (fails for tests outside the given tests). For each vulnerability (each row of Table 7.4), we created a test suite T_m which includes the failing test case and a passing (non-crashing) test case. Then, we give T_m and the fix location to $F1X$ for patch generation. In summary, $F1X$ was able to fix 19 bugs out of 24, but only eight out of the 19 fixes are not overfitting (as shown in rows with x/0 in the "Fuzz" column in Table 7.6). Effectively, only eight bugs have been successfully repaired by $F1X$.

Prophet [79]: We also evaluate our benchmark with Prophet, one of the popular search-based automated repair tools; We choose to evaluate on Prophet because it represents a state-of-the-art program repair tool which has been shown in [79] to be more efficient and effective than other search based tools. Prophet uses a probabilistic, application-independent model generated from a collection of human written patches, to find the correct code. We run Prophet using the default configuration (with the pre-trained model and enabled the condition-ext option and the replace-ext option). For each vulnerability (each row of Table 7.4), we created a test suite T_m which includes the failing test case and a passing (non-crashing) test case. Then, we give T_m and fix location to Prophet, to repair the bug. In summary, prophet was able to repair 12 bugs out of 24, however only three of the 12 fixes are not overfitting. Effectively, only three bugs have been successfully repaired by Prophet.

Compared to Prophet, F1X was able to generate higher number of correct patches for our benchmark. Prophet is a search-based technique which modifies the code and rely on the test suite for correctness of the code, whereas F1X generates a patch using the constraints generated by the test suite. Hence, the semantic-based repair approach is more effective than the search-based repair approach in our experiments.

Although $F1X$ and Prophet was able to fix most of the bugs listed in our benchmark, the quality of the patches are low. This is indicated in Table 7.6 column "$F1X$" and "Prophet" with the patches produced by both failing on a large number of differential fuzzing test cases, where the output generated by the program before

```
 static void j2k_read_siz(...)
 {

- cio_read(cio, 2);
+ if (0)
+   cio_read(cio, 2);

}
```

(a) F1X Patch

```
 static void j2k_read_siz(...)
 {
- if (e->id == id) {
-    break;
+ if (e->id == id && !(1) ) {
+    break;

}
```

(b) Prophet Patch

```
 static void j2k_read_siz(...)
 {
+ if (image->comps[i].dx == 0 ||
       image->comps[i].dx > 255) {
+       return;
+   }

}
```

(c) Transplanted Patch

```
 static void j2k_read_siz(...)
 {
+ if (!(image->comps[i].dx *
       image->comps[i].dy)) {
+       opj_event_msg(..);
+       return;
+   }
}
```

(d) Developer Patch

Figure 7.14: Comparison of transplantation vs APR patch for bug-1

the fix and after the fix differ for non-crashing instances. Figure 7.14 shows the patch comparison for bug ID 1 in Table 7.4. F1X generates an overfitting patch for the divide by zero bug (i.e., bug ID 1 in Table 7.4), which avoids the execution of an API call by inserting a condition which is always false. The quality of the patch is determined by the differential fuzz testing of the patch, which shows failure for 13 out of 100 test cases generated using fuzzing, as indicated in Table 7.6. Similarly, Prophet generates a patch which exclude the execution of a statement by inserting a false logical connective. The quality of the patch is revealed to be over-fitting by the differential fuzz testing of the patch which shows failure for two test cases on average for 100 generated test cases. However, the transplanted patch which is adapted from a developer patch from JasPer program does not fail on any of the fuzz input generated for differential analysis. Moreover, manual inspection of the patch, as shown in Figure 7.14d is correct (semantically equivalent to human patches) for the divide by zero bug, which checks if the denominator is zero. Furthermore, our evaluation also emphasizes the following: (1) quality of the patch in automated repair tools depends on the quality of the test suite, (2) it is difficult for automated repair tools to produce multi-line fixes. Compared to human written patches current methods produce lower quality patches [37], hence augmenting transplantation techniques can improve the quality of the patches.

7.6.2.5 Comparison with Transplantation Tools

We compare our tool with existing transplantation tools to investigate the effectiveness of our approach. There are several transplantation tools proposed by prior work. CodePhage [126] is the most relevant tool for comparison as it transfers security fixes, compared to μSCALPEL [9] and CodeCarbonCopy [125] which transfers functionality. However, both CodePhage and CodeCarbonCopy are not publicly available for evaluation. We requested the authors of CodePhage [126] to provide access to the tool for comparison purpose, however due to unavoidable circumstances the primary developer was not available, hence we were not able to obtain the tool. Since μSCALPEL is the only publicly available tool for evaluation in C programs, we evaluate our approach against μSCALPEL [9]. Table 7.6.2.5 gives an overview of the comparison.

We compare PatchWeave with μSCALPEL [9] for four errors, since it is designed for transplantation of a feature. We only consider comparing PatchWeave with μSCALPEL for patches that involves transplantation of a new function. Specifically, we manually specify the insertion point for the patch and evaluate the effectiveness of μSCALPEL for four errors. Note that μSCALPEL has an advantage over PatchWeave under this setup because it does not need to search for the entry points and the insertion points. For each of these errors, as μSCALPEL is based on genetic programming (GP), we rerun μSCALPEL for 10 times with different seeds for 30 minutes for each run due the stochastic nature of GP. Table 7.6.2.5 shows that all of the runs for μSCALPEL could not complete as they resulted in segmentation fault. The reason is because the implementation of μSCALPEL was unable to parse the function in the donor program specifically "JasPer" which is the donor program for the four errors we tried for transplantation. μSCALPEL fails to extract the function from the donor program hence unable to successfully repair the bug. We have reported this issue to the developers of μSCALPEL [9] and the developers have acknowledged the issue. Similar inefficiency of μSCALPEL confirmed with the results of another prior experiments of μSCALPEL that have been independently conducted by other researchers [82].

Table 7.7: PatchWeave comparison with Transplantation Techniques

ID	Donor	Target	Error	Patch Class		μSCALPEL
1	JasPer-1.900.3	OpenJPEG-1.5.1	DZ	Class-III	✓	N/A
2	OpenJPEG-1.5.2	JasPer-1.900.2	DZ	Class-III	✓	N/A
3	JasPer-1.900.13	OpenJPEG-1.5.1	IO	Class-IV	✓	Seg Fault
4	OpenJPEG-1.5.2	JasPer-1.900.12	IO	Class-III	✓	N/A
5	JasPer-2.0.12	OpenJPEG-1.5.1	NPD	Class-III	✗	N/A
6	OpenJPEG-1.5.2	JasPer-1.900.30	NPD	Class-III	✓	N/A
7	JasPer-1.900.3	OpenJPEG-1.3	DZ	Class-III	✓	N/A
8	OpenJPEG-1.4	JasPer-1.900.2	DZ	Class-III	✓	N/A
9	JasPer-1.900.14	OpenJPEG-2.1.0	UIO	Class-III	✓	N/A
10	OpenJPEG-2.1.1	JasPer-1.900.13	UIO	Class-III	✓	N/A
11	OpenJPEG-1.5.1	LibWebP@0.3.0	SIO	Class-III	✓	N/A
12	JasPer-1.900.4	LibWebP@0.2.0	ShO	Class-II	✓	Seg Fault
13	LibWebP@0.3.0	JasPer-1.900.3	ShO	Class-I	✓	N/A
14	OpenJPEG-1.5.2	JasPer-1.900.13	HBO	Class-III	✗	N/A
15	JasPer-1.900.14	OpenJPEG-2.1.0	HBO	Class-IV	✗	Seg Fault
16	JasPer-1.900.17	OpenJPEG-1.5.2	UIO	Class-III	✓	N/A
17	LibTiff-3.8.0	Jasper-1.900.8	MWE	Class-III	✓	N/A
18	Libsndfile-1.0.26	WavPack-5.1.0	ShO	Class-III	✗	N/A
19	Jasper-1.900.13	Jasper-1.900.2	IO	Class-II	✓	Seg Fault
20	LibMing-0.4.8	LibMing-0.4.6	DZ	Class-I	✓	N/A
21	LibMing-0.4.8	LibMing-0.4.6	NS	Class-I	✓	N/A
22	LibTiff-4.0.8	LibTiff-4.0.0	HBO	Class-I	✓	N/A
23	Libsndfile-master	Libsndfile-1.0.26	GBO	Class-I	✓	N/A
24	Libzip-1.3.0	Libzip-1.1.2	MAF	Class-I	✓	N/A
	Total			24	20	0

Division By Zero Error {DZ - Divide by Zero}, *Integer Overflow* {IO - Integer Overflow, UIO - Unsigned Integer Overflow, SIO - Signed Integer Overflow}, *Memory Error* {NPD - Null Pointer Dereference, MAF - Memory Allocation Failure, MWE - Memory Write Error}, *Shift Overflow* {ShO - Shift Overflow, NS - Negative Shift}, *Buffer Overflow* {HBO - Heap Buffer Overflow, GBO - Global Buffer Overflow}

7.6.2.6 Syntactic vs Semantic Patch Transplantation

While PatchWeave performs semantic patch transplantation based on concolic execution, existing approach (i.e., LASE [93]) infers syntactic edit scripts from examples (in our problem formulation in Figure 7.1, P_b serves as one such example) and uses the inferred scripts to find edit locations, customizes the script to each location, and applies the customized script. Since LASE targets Java programs and there are no other syntactic patch transplantation tools available for C programs, we implemented a prototype inspired by the LASE [93] technique for comparison purpose. Our comparison tool uses the same technique of clone detection and AST transformation to locate the insertion point and transplant the patch. This allows us to perform a comparison of syntactic vs semantic approaches for patch transplantation. Our prototype implementation uses clone detection to identify a similar function in P_c with the help of Deckard [55] syntactic distance calculation. It uses GumTree [34] algorithm to formulate an AST transformation script which can be used to transplant the patch from P_b to P_c. Table 7.8 shows the overall results of the comparison. The "Time" column shows the total time taken in minutes for each tool to fix the error/vulnerability, or ✗ to indicate the repair was unsuccessful. The "Fuzz" column denotes the results of differential fuzz testing in the form of x/y, where x is the number of test cases where P_c results in a crash and P_d gracefully exits, whereas y is the number of test case where P_d crashes or produce a different output than P_c. The "Function" column indicates that if the syntactic approach was able to correctly identify the insertion function for the transplantation, and the "Var Map" column represents if the AST node matching for variable mapping based on GumTree [34] algorithm was able to correctly map variables used for the transplantation.

Syntactic approach was able to successfully repair almost all bugs in Class-I and Class-II (six out of eight bugs) because these bugs do not require variable translations. The donor fix included changes which are not relevant for the bug in ID 13, hence the syntactic approach failed to filter the relevant fix. These two classes represent syntactically very similar programs such as backporting versions or forked projects, hence AST node matching could find the correct insertion point once the inserting function is located using clone detection. One interesting observation in our experiment is that for Class-I and Class-II syntactic method performs better compared to the semantic method. This is because the semantic method requires expensive symbolic execution to calculate the insertion point while syntactic method only requires clone detection which is relatively lightweight. Syntactic method was not able to transplant any of the fixes that require a translation of variables although in few cases it was able to identify the correct insertion function. This is due to the failure of syntactic approach to map variables across different data-structures. In these cases, our approach PatchWeave is more effective.

Table 7.8: PatchWeave comparison with Syntactic Patch Transplantation

ID	Donor	Target	Error	Patch Class	Semantic Method		Syntactic Method			
					Time	Fuzz	Function	Var Map	Time	Fuzz
1	JasPer-1.900.3	OpenJPEG-1.5.1	DZ	Class-III	5.0	37/0	✓	✗	✗	-/-
2	OpenJPEG-1.5.2	JasPer-1.900.2	DZ	Class-III	8.0	22/0	✗	✗	✗	-/-
3	JasPer-1.900.13	OpenJPEG-1.5.1	IO	Class-IV	7.5	5/0	✓	✗	✗	-/-
4	OpenJPEG-1.5.2	JasPer-1.900.12	IO	Class-III	8.5	42/0	✗	✗	✗	-/-
5	JasPer-2.0.12	OpenJPEG-1.5.1	NPD	Class-III	✗	-/-	✗	✗	✗	-/-
6	OpenJPEG-1.5.2	JasPer-1.900.30	NPD	Class-III	15.0	9/0	✗	✗	✗	-/-
7	JasPer-1.900.3	OpenJPEG-1.3	DZ	Class-III	3.5	52/0	✗	✗	✗	-/-
8	OpenJPEG-1.4	JasPer-1.900.2	DZ	Class-III	5.5	20/0	✓	✗	✗	-/-
9	JasPer-1.900.14	OpenJPEG-2.1.0	UIO	Class-III	6.0	42/0	✗	✗	✗	-/-
10	OpenJPEG-2.1.1	JasPer-1.900.13	UIO	Class-III	18.0	46/0	✗	✗	✗	-/-
11	OpenJPEG-1.5.1	LibWebP@0.3.0	SIO	Class-III	2.5	90/0	✗	✗	✗	-/-
12	JasPer-1.900.4	LibWebP@0.2.0	ShO	Class-II	5.0	99/0	✓	✓	4.0	99/0
13	LibWebP@0.3.0	JasPer-1.900.3	ShO	Class-I	11.0	98/0	✓	✓	✗	-/-
14	OpenJPEG-1.5.2	JasPer-1.900.13	HBO	Class-III	✗	-/-	✗	✗	✗	-/-
15	JasPer-1.900.14	OpenJPEG-2.1.0	HBO	Class-IV	✗	-/-	✗	✗	✗	-/-
16	JasPer-1.900.17	OpenJPEG-1.5.2	UIO	Class-III	12.0	42/0	✓	✗	✗	-/-
17	LibTiff-3.8.0	Jasper-1.900.8	MWE	Class-III	26.0	18/0	✗	✗	✗	-/-
18	Libsndfile-1.0.26	WavPack-5.1.0	ShO	Class-III	✗	-/-	✗	✗	✗	-/-
19	Jasper-1.900.13	Jasper-1.900.2	IO	Class-II	8.0	34/0	✓	✓	5.5	40/0
20	LibMing-0.4.8	LibMing-0.4.6	DZ	Class-I	5.5	98/0	✓	✓	10.5	100/0
21	LibMing-0.4.8	LibMing-0.4.6	NS	Class-I	5.5	29/0	✓	✓	6.0	32/1
22	LibTiff-4.0.8	LibTiff-4.0.0	HBO	Class-I	26.0	75/0	✓	✓	8.0	76/1
23	Libsndfile-master	Libsndfile-1.0.26	GBO	Class-I	16.0	73/0	✗	✗	✗	-/-
24	Libzip-1.3.0	Libzip-1.1.2	MAF	Class-I	42.0	90/0	✓	✓	6.0	95/0
	Total			24	20	20	11	7	6	4

Division By Zero Error {DZ - Divide by Zero}, *Integer Overflow* {IO - Integer Overflow, UIO - Unsigned Integer Overflow, SIO - Signed Integer Overflow}, *Memory Error* {NPD - Null Pointer Dereference, MAF - Memory Allocation Failure, MWE - Memory Write Error}, *Shift Overflow* {ShO - Shift Overflow, NS - Negative Shift}, *Buffer Overflow* {HBO - Heap Buffer Overflow, GBO - Global Buffer Overflow}

7.6.3 Threats to Validity

There are several threats to validity of our approach, related to the external tools we use or the datasets that we use in our experiments. We seek to mitigate such threats by using actively maintained external tools such as KLEE, and conducting our experiments on a wide variety of subjects, including those studied by previous work on software transplantation [126].

We would like to highlight two issues related to the availability of tests. First, we assume the presence of at least one test (i.e., the exploit). If no test is available, as may be the case in certain domains (e.g., backporting of Linux patches [103]), our technique cannot be applied as it is. Furthermore, relying only on one test case could result in a patch that is overfitted, which is likely to disrupt correct behavior of the application. We address this concern, with the use of differential fuzz testing where we generate mutated inputs from the single failing test case. Then we employ differential behavior analysis to detect if the transplanted patch introduces disruption to correct behavior, by comparing the output of P_c and P_d for each generated test-case. Second, due to practical considerations, we have not assumed the presence of a large number of tests. Moreover, requiring more than one test case is often an impractical requirement since more often vulnerability reports contain a single-exploit, hence our focus is to provide an automation to help developers fix the issue given a single test-case is provided. If more tests are available, we could extend our technique to efficiently navigate all candidate patches and select one (similar to the selection of patches in prior work in program repair e.g. F1X [89]). However, such a patch space exploration is *not* part of our current PatchWeave implementation.

7.7 Summary

In this chapter we formulate the *patch transplantation* problem. We also propose a fully automated solution to patch transplantation based on concolic execution. The patch transplantation problem caters to a *real-life need* in the practice of software security: even when an important vulnerability is detected and a patch is constructed, it is non-trivial to adapt the patch for other similar implementations exhibiting the same vulnerability. Indeed, associating a CVE to a newly found vulnerability and publishing the patch, may make these other similar un-patched implementations more vulnerable since attackers will have more knowledge on how to exploit the vulnerability. The patch transplantation problem studied in this paper provides a solution to reduce or remove such exposure to vulnerabilities.

Chapter 8

Related Work

This chapter discusses the existing related work in the areas of program repair, software transplantation, symbolic execution, program synthesis and security vulnerability repair. We summarize the related work and discuss the limitations.

8.1 Automated Program Repair

Automated program repair [97] is an emerging technology, which seeks to automatically rectify program errors, typically as observed via failure of tests or assertions. Several approaches have been proposed to automatically generate patches [100, 79, 154, 160, 92, 138, 88, 105, 91, 68, 161, 158, 136, 146]. A review of the area appears in [69]. Common techniques for automated repair include program mutations via genetic search [68], specification inference via symbolic execution or SAT solving [45, 100, 92], repair via abstract interpretation [76], code transplantation [126], and learning and prioritization of patch candidates and fix patterns [6, 80, 79, 116]. CPR is more related to specification inference based program repair. These approaches employ symbolic execution to generate a repair constraint, which the buggy program needs to satisfy to pass a given test-suite. Solutions to the repair constraint, in the form of patch expressions, are then obtained using program synthesis. Most of the existing works on test-based program repair suffer from test data overfitting, where the patched program fails for tests outside the given test-suite [69, 110]. To alleviate overfitting, one may use more general oracles beyond tests [36], or may generate tests to rule out overfitting patches [39]. Certain works develop customized repair strategies for fixing security vulnerabilities by either employing heuristics [52], by applying fix templates that avoid specific errors [127], or by hooking up with sanitizers [40]. In contrast, CPR is a general purpose repair engine, though we have also shown its efficacy on the dataset of [40]. CPR generates tests from an initial seed test by modifying the path condition, in the style of concolic execution. However, the path of a test contains yet to be inserted patches. Hence the path exploration in concolic execution is accompanied by a systematic reduction of the pool of patch candidates in CPR. Finally, counterexample-guided inductive synthesis (CEGIS) [134, 135, 4] represents a synthesis technique, in which the desired solution is iteratively refined based on a loop between a *generator* and a *verifier*. CPR also leverages counterexamples to reduce the patch space, and has some relationship to CEGIS. In CPR, we use a counterexample-guided refinement of the parameter constraints of the available patches. The work of [64] performs concolic execution on specific tests to check whether a patch candidate meets a specification; if it does not, the resultant constraint is added for the generation of future repair candidates. In contrast, CPR works on abstract patch candidates and refines them. Furthermore, [64] terminates as soon as there is no counterexample anymore, which again can lead to functionality deleting patches.

Furthermore, the work of [90] uses unrealizability of patch expression synthesis for pruning paths in symbolic analysis. The recent work of [51] also studies unrealizability of syntax guided synthesis problems. In comparison, CPR use of concolic path exploration for program repair also prunes feasible paths, where none of the patch expressions from the remaining patch pool can be inserted. For scalability of reasoning we do not maintain the patches as second order variables; instead they are maintained as abstract patch templates with parameters denoting interval constraints.

8.2 Software Transplantation

Automated software transplantation has been applied for solving several software maintenance tasks, including feature transplantation [9, 86], transplanting validation checks [126], and transplanting shellcode for remote exploits [8]. μSCALPEL [9] and CodeCarbonCopy [125] transplant new functionality from a donor application P_b into a recipient application P_c. Both of these approaches require a developer to specify the insertion point and identify the functionality to be extracted for transplantation. μSCALPEL [9] uses genetic programming with program slicing to transplant functionality from a donor system to a target system but requires manually specifying the entry point of code and the insertion point in the host program. Meanwhile, CodeCarbonCopy [125] requires manual specification of (1) the donor function that captures the functionality to be transplanted, (2) the insertion point in the host program, and (3) extra parameters for removing irrelevant functionality in the transferred code. Code Phage [126] eliminates errors such as integer overflow by transferring checks from P_b to P_c. All three approaches are limited in transplanting new code (e.g., function and check condition) and could not handle the case where the donor and the recipient are code edits in the form of a patch. Meanwhile, PatchWeave provides a fully automated approach for patch transplantation without the manual effort required to specify the donor function and the insertion point.

Several techniques transplant Java code edits [166, 82]. Although the donor P_a and the recipient P_c in the patch transplantation problem could be regarded as code clones, GRAFTER [166] uses code clones for test reuse and differential testing, whereas PatchWeave uses the similarity between P_a and P_c to automatically identify insertion points. Similar to PatchWeave that encourages code reuse, program splicing [82] reuses existing code from the web, whereas we reuse security patches. Overall, [166] and [82] supports Java programs instead of C programs but prior study shows that there are less exact clones in C functions than Java methods [114]. The lack of exact clones in C functions implies that the clones' adaptation in C programs could be more challenging than in Java programs.

8.3 Patch Backporting

To help developers backport patches, several approaches have been proposed [112, 144]. Tian et al. [144] proposed an approach to automatically identify bug-fixing patches that should be backported to old versions. Ray et al. [112] proposed to detect and characterize porting errors to help developers avoid them. In contrast, we directly backport patches and provide patch suggestions for developers. Another line of relevant work is the Backports Project [5], which enables old Linux kernels to run the latest drivers. The Backports Project develops a set of tools to automate the backporting process for Linux drivers [102, 143] to make them compilable with old kernel versions. The Backports Project uses the program matching and transformation tool Coccinelle [65] to allow developers to express backporting transformation in a generic way that is expected to be applicable to many versions. In contrast, FixMorph is fully automated and does not require manually created transformation rules. A prior approach by Thung et al. [143] automatically extracts code transformation rules. However, this approach requires guidance from compilation errors, and it can only transform patches that affect a single line of code.

Program transformation is a similar line of work which infer transformation rules from human-written patches and transfer patches to another codebase by applying the inferred rules. Program transformation has been applied to many software maintenance tasks, including automating repetitive code edits [113, 94, 93, 96, 99], intelligent refactoring [95, 38] and fixing software bugs [10, 77, 6]. Those approaches solve problems similar to FixMorph, but there are key differences. Most existing works infer transformation rules from multiple human patches, while FixMorph synthesizes rules from only one patch. Although GenPat [54] and Sydit [94] also rely on only one example, they either require a large codebase to provide statistical information or synthesize rules by simply generalizing all identifiers, which results in many false positives, as also confirmed by our experiments on Linux. Second, some existing transformation techniques [10, 77, 6] transform patches across different projects, and the others [95, 38, 93, 94] transform patches within the same codebase. In contrast, FixMorph leverages the similarity between Linux kernel versions to synthesize properly generalized transformation rules.

8.4 Trust Aspects in APR

Related work includes considerations of trust issues [115, 3, 11] and studies about the human aspects in automated program repair [17, 139, 71, 37, 59], user studies about debugging [104], and empirical studies about repair techniques [74, 63, 98, 151, 156, 163, 87, 73].With regard to human aspects in automated program repair, our survey study contributes novel insights about the developers' expectations on their interaction with APR and which mechanism help to increase trust. With regard to empirical studies, our evaluation contributes a fresh perspective on existing

APR techniques.

Trust issues in automated program repair emerge from the general trust issues in *automation*. Lee and See [70] discuss that users tend to reject automation techniques whenever they do not trust them. Therefore, for the successful deployment of automated program repair in practice, it will be essential to focus on its human aspects. With respect to this, our presented survey contributes to the knowledge base of how developers want to interact with repair techniques, and what makes them trustworthy.

Existing research on trust issues in APR focuses mainly on the effect of patch provenance, i.e., the source of the patch. Ryan and Alarcon et al. [115, 3] performed user studies, in which they asked developers to rate the trustworthiness of patches, while the researchers varied the source of the patches. Their observations indicate that human-written patches receive a higher degree of trust than machine-generated patches. Bertram et al. [11] conducted an eye-tracking study to investigate the effect of patch provenance. They confirm a difference between human-written and machine-generated patches and observe that the participants prefer human-written patches in terms of readability and coding style. Our study, on the other hand, explores the expectations and requirements of developers for trustworthy APR. The work of [153] proposed strategies to assess repaired programs to increase human trust. Our study results confirm that an efficient patch assessment is crucial and desired by the developers. We note that [153] focuses on how to assess APR, while we focus on how to enhance/improve APR in general, specifically in terms of its trust.

8.5 Patch Validation

Since G&V techniques enumerate a search-space to generate patches and test each candidate patch against a provided test oracle, they scale to relatively small search spaces. In order to achieve effectiveness for a large search-space several works on optimizing the repair process have been discussed in the literature. Optimizations for G&V techniques have been studied widely over recent years to improve the quality of the patches [138, 39] and to improve performance efficiency [89, 154]. GenProg-AE [154] improves overall repair time by reducing validation cost spent on redundant test executions by identifying program-equivalence of functionality via lightweight analysis.

F1X [89] reduces validation cost using program-equivalence over the space of candidate patches. Various equivalence relations can be accommodated into the framework based on values or program dependencies. Such an approach works on an *implicit* representation of the patch space. Such program-equivalence relation based optimizations are orthogonal to our compilation-free repair approach, as we reduce the cost of recompilation while traversing an explicitly represented search space of edits. The focus of our work is to improve generate and validate search based repair tools, tools which work with an explicit representation of the search space.

More recent work to address the recompilation cost have been proposed using on-the-fly patch validation [41, 23]. PraPR [41] uses mutation at the byte-code level to achieve on-the-fly validation for JVM based program repair. Extracting the byte-code changes and directly applying the mutation at run-time is shown to be an order of magnitude faster than state-of-the-art for JVM based byte-code repair tools. In combination with HotSwap technique, PraPR [41] can bypass expensive compilation on patch generation, process creation and JVM warm-up in order to validate a candidate patch. However PraPR is limited to only repairs that can be fixed via byte-code manipulation and is shown to be imprecise [23]. UniAPR [23] extends the work of PraPR to source-code and byte-code for JVM based repair techniques. Compared to PraPR, UniAPR provides on-the-fly patch validation to source-level by translating the patch into byte-code changes and reusing PraPR for validation. UniAPR does not completely eliminate the recompilation cost since it uses incremental compilation to obtain the byte-code changes that can be injected at runtime. In comparison, we propose to completely remove the compiler from the repair-loop by using an interpreter to validate the patches on-the-fly. CFR is not limited to specific language or repair tool, in fact our technique can be applied for Java repair tools provided a lightweight interpreter such as the GDB, is tuned for Java.

In order to improve the latency of program repair several approaches have been proposed to prioritize the search-space. Prophet utilizes features extracted from human-written patches to predict a correctness score for each candidate patch and enumerate the search-space in the ranked order to effectively find the correct/-plausible patch in less amount of time. ACS [160] prioritizes patches based on information gained from documentation of the source-code for older versions of the software, while S3 [66] use a combination of semantic and syntactic properties to prioritize its search space. Such optimizations are synergistic to CFR, and can be achieved on top of the efficiency derived from compilation-free repair.

Chapter 9

Conclusion

This chapter summarizes the results and contributions of this thesis work, discusses our perspectives and the future work.

9.1 Summary and Impact

In this thesis, we have investigated the challenges in generating security patches to preserve security of software systems and proposed several techniques to automate current processes, to reduce the time to fix thereby reducing the exposure window of identified software security vulnerabilities. We first studied the impeding challenges in the adoption of existing program repair tools and the trustworthiness of auto-generated patches perceived by developers. Identifying the limitations and challenges in existing program repair tools helps to shape the future work on program repair to the extent of wider adoption of program repair in practice. Considering the insights gained from our study, we propose a solution to speedup the efficiency of generate & validate repair techniques, that replace the compiler from the repair-loop with a lightweight interpreter that is able to achieve the same result at a low cost. Using on-the-fly patch validation with existing program repair tools for C/C++ programs we show the benefits of compilation-free repair. This work also shows the benefit of a concurrent repair framework that integrates multiple repair tools with a single validation back-end to improve throughput and latency in program repair. In addition, to provide correctness guarantees, we proposed a novel program repair technique “concolic program repair” that efficiently navigates a large-scale search-space for patch generation, while ensuring high-quality patches are generated with respect to a user-provided program specification. In doing so, we also provide additional guarantees for the correctness of the generated patches by generating new test-cases. Moreover, we study two additional problems related to fixing software security vulnerabilities; namely the automated patch backporting problem and the automated patch transplantation problem. A trustworthy patch that is generated for the reported software security vulnerability can be backported to various other older versions of the software, using our proposed backporting technique. Moreover, the same patch can also be transplanted to other semantically equivalent programs which may exhibit potential for a similar variant of the identified software security vulnerability. At it's core, our contributions work cohesively to alleviate the problem of generating security patches.

9.2 Perspectives

9.2.1 Importance of Vulnerability Repair

A recent study on software security vulnerabilities reveals that 2021 alone there have been 28,695 flaws disclosed which is a record breaking amount with a significant increase from 23,269 in 2020 [118]. Additionally, the report finds the average time to fix vulnerabilities (i.e. window of exposure) is 200 days and for high-severity vulnerabilities its 246 days, which also aligns with our findings in our Linux empirical study. Despite empirical studies providing insights and evidence of the increasing importance of vulnerability repair especially of those in software, there is little to no work to improve the state of the art in reducing the impact of such vulnerabilities for the average end-user. Our empirical analysis on the Linux kernel project indicates alarming delays in providing necessary fixes to identified vulnerable systems, despite the availability of a solution. The current patch propagation mechanism are inadequate to provide a solution to all affected systems (i.e. fixing all vulnerable stable versions), in some cases as observed from the Linux back-porting effort may not have a fix provided resulting in a vulnerabilities open for exploitation for several months. Recently disclosed vulnerability Log4J [49] further reveals the inefficiency in practice to systematically identify and provide timely fixes to mitigate the threat from been exploited by malicious users. In fact, Log4J [49] is an example which highlights the efficiency of the malicious actors in leveraging the information of publicly disclosed information to generate an exploit compared to developers effort to generate a fix. Our investigations into patch backporting and patch transplantation should call for more research in this direction, to facilitate efficient solutions in addressing the challenges faced by developers to keep software secure. More focus should be given to reduce the average time to fix critical and high severity vulnerabilities to improve the window of exposure and consequently the overall security posture of applications.

9.2.2 Automation is the Key

Given the importance of fixing software vulnerabilities and the increasing numbers of malware exploiting such vulnerabilities, it is crucial to invest on automation and improve the infrastructure to facilitate the demands of generating effective solutions much more efficiently. Automated Program Repair (APR) is one such potential solution to alleviate the burden of generating patches by assisting the developers to find quick solutions. However, one of the main challenges in APR (as with any automation technology) is the trust for the auto-generated patches perceived by the developers, who will ultimately decide if the generated patch is suitable to be applied/integrated. While previous work highlighted the bias in developers and the inertia to adopt automated solutions, we tried to identify how to enhance the trust by developers as a step towards achieving the goal of wide scale adoption of APR. Our empirical study consisting of 100+ software practitioners is one of the first large-scale empirical study in program repair that shows positive response for APR adoption. We also capture the challenges and requirements expected by the developer in order to achieve adoption that gives direction to the APR community when designing such systems. Overall there was a positive response from the developers to adopt APR into their development processes, given the necessary trust requirements are met. Although our empirical study focused on automated program repair, the survey methodology and result interpretation code-book can be inspiration on conducting empirical analysis for other automation technology as well. The focus of our study was to capture the perception for a specific technology (i.e. APR) and gain insights for the requirements for wide scale adoption.

9.2.3 Compilation-Free Repair

Current state of the art APR techniques mostly focused on finding the correct solution in an unconstrained environment. However, the need in practice enforces several constraints on the repair techniques such as low latency and high throughput. If APR is to be adopted in practice, it should be able to enumerate a large search-space efficiently and generate a ranked list of plausible patches within a short duration. We investigated one commonality in the repair techniques that can be improved, which is the patch validation step. Patch validation is one of the steps in program repair specifically for generate & validate techniques whereby candidate patches are validated using a suitable test oracle. A key step for validation is re-compilation which translates the candidate patch into an executable binary that can be tested against a test oracle. In our work on compilation free program repair we show significant performance gains, irrespective of the search strategy used to navigate the explicitly defined search space of patch candidates. Of course how much performance gain is accrued may differ, based on the search strategy. Our proposed technology can open new research directions for program repair taking advantages of interoperability within existing repair systems. Exchanging information between several repair techniques could lead for high-quality patches, where the fix ingredients are generated by different repair techniques combined with our integrated framework. In addition to improvement in space size, a divide-and-conquer approach can improve the search where each repair tool enumerates a separate partition of the search space simultaneously. Generating multi-line patches becomes viable if there is an increase in patch space exploration speed allowing repair tools to navigate a much richer patch space. Existing repair techniques need to tackle the problem of combinatorial explosion of the search space when generating multi-line patches, which can be alleviated by an efficient patch-validation technique such as compilation-free repair. In addition to multi-line repair it will be interesting to incorporate multi-language repair which combines repair technology of different languages such as C language and Java language for software systems that requires code from both languages. Such repair would benefit software systems where the library is developed using a low-level language such as C and the application is developed using a high-level language such as Java.

9.2.4 Gradual Correctness

Another significant difficulty in program repair comes from the lack of *complete* specification of intended program behavior. Since a detailed specification of correct behavior is usually not available, existing program repair techniques are guided by tests. Relying on test cases has lead to the patch overfitting problem, which effects the reliability of the generated patches leading to trust issues perceived by developers. Lack of specification affects the repair problem in the initial stage of understanding the correct behavior to fix the program and in the latter stage of providing guarantees to the developer for the correctness of the patch. In our work on "concolic program repair" we relaxed the requirement of *complete* specification into *partial* specification which is a tractable problem compared to inferring whole program specification and providing verification for such. Especially for software vulnerabilities a crash free constraint can serve the purpose of a specification to summarize the correct behavior with respect to a given security property, similar to a developer provided program assertion capturing the logical behavior of the program. This fresh look into the specification problem allowed us to provide guidance into program repair such that it can be served as a means of detecting and discarding overfitting patches that only pass the provided test-suite. Furthermore, such specification can also be used to generate new test-cases to provide additional observations for the correctness of the patch that enhance the trust perceived by the developer, since additional test-cases was an indication for correctness reported by developers in our survey. Our repair technique is the first of its kind that can incorporate a user-provided specification into the repair process and generate additional verification for the generated patches. Conceptually, we also presented a viewpoint of "*gradual correctness*", where the repair algorithm was formulated as an anytime algorithm that systematically explored both the input space and patch space. This notion of gradual correctness, can also be meaningful for program synthesis, recovery and transplantation. Gradual correctness can thus help us produce high quality automatically constructed code.

9.2.5 Recurring Vulnerabilities

Generating a patch for the identified software vulnerability alone is not sufficient to mitigate the threat exposed by the discovery of the vulnerability. The generated repair should be propagated to all versions of the software that has been impacted by the discovery. To understand the effect and challenges in providing automated solutions to propagate such patches, we investigated the backporting activities in the Linux kernel because it is a large-scale widely used codebase. The sheer complexity of the patches, the diversity of the transformations involved, and the absence of test cases as specification pose additional challenges for patch backporting. Due to the popularity and importance of the Linux kernel, it could be worthwhile for the program repair community to evaluate efficacy of repair techniques on Linux as well. In our envisioned workflow, APR will generate the initial fix, and it can be automatically backported into an old stable versions. Instead of trying to generate fixes for each version separately, the workflow should reuse the information from one patch to generate a fix into other program versions. FixMorph shows the promise of automated backporting of patches on the Linux kernel code-base, thereby demonstrating the practical promise of such techniques. Apart from reducing exposure to security vulnerabilities, such patch backporting is of significant practical value for automating software maintenance tasks.

We also formulated the *patch transplantation* problem in this thesis. Which considers the impact of similar software, which extends the notion of similarity into higher order abstractions such as standards, protocols etc. We proposed a fully automated solution for patch transplantation based on concolic execution, to leverage the similarity and generating fixes based on a repair from a semantically similar program. The patch transplantation problem caters to a *real-life need* in the practice of software security: even when an important vulnerability is detected and a patch is constructed, it is non-trivial to adapt the patch for other similar implementations exhibiting the same vulnerability. Associating a CVE to a newly found vulnerability and publishing the patch, may make these other similar unpatched implementations more vulnerable since attackers will have more knowledge on how to exploit the vulnerability. The patch transplantation problem studied in this thesis provides a solution to reduce or remove exposure to such vulnerabilities.

9.3 Future Work

Our research work reveals several interesting future research directions. Although we proposed several techniques to alleviate the problem of automatic patching of software security issues, automatically fixing software security vulnerability is still a problem that is not completely solved, and can have a significant financial/economical impact. Current state of the art for program repair are still in the early stage of both research and practice. Reflecting on our findings and perspectives, we envision the following important future research work.

9.3.1 Trustworthy Program Repair

One of the key-challenge for the program-repair community to address is to gain developer trust in auto-generated patches. In our preliminary investigations into trust enhancement issues of program repair reveals current state-of-the-art techniques do not meet the requirements of the software developers/maintainers. Specifically, providing additional guarantees and explanation for the generated repair, are some of the key-aspects that should be investigated. Moreover, developers require the exchange of artifacts such as generated tests as inputs as well as output of repair tools. Further study is required to improve the bias developers would associate with an automated solution and how to overcome such bias by enhancing the trust for the auto-generated patches. More empirical analysis with controlled user-studies could help understand the challenges in realizing practical APR. For instance, an empirical evaluation of existing program repair tools with software practitioners using the tools in their day-to-day development activities could provide better insights on the challenges to adopt APR in practice.

9.3.2 Program Repair for Binaries

The proposed techniques in this thesis, works at the source-level, where security patches can be efficiently generated. First and foremost, it would be interesting to take forward the techniques proposed in this work for binary repair. Front-line defense against exploitation of software security vulnerabilities remains at the binary executable that are deployed in software systems. Transplanting patches into binary executable can further increase the security of software systems. These patches can be generated from either source-level program-repair techniques or manually crafted patches by the developer. Emergence of binary-rewriting techniques [30] shows promise in the direction of repairing program binaries. Integrating our proposed approaches with program binaries can minimize the impact of software security vulnerabilities.

Bibliography

[1] M. D. Adams and F. Kossentini, "Jasper: A software-based jpeg-2000 codec implementation", in *Proceedings 2000 International Conference on Image Processing (Cat. No. 00CH37101)*, IEEE, vol. 2, 2000, pp. 53–56.

[2] "Agostino's blog", https://blogs.gentoo.org/ago, Accessed: 2019-02-24, 2019.

[3] G. M. Alarcon, C. Walter, A. M. Gibson, R. F. Gamble, A. Capiola, S. A. Jessup, and T. J. Ryan, "Would you fix this code for me? effects of repair source and commenting on trust in code repair", *Systems*, vol. 8, no. 1, 2020, ISSN: 2079-8954. [Online]. Available: https://www.mdpi.com/2079-8954/8/1/8.

[4] R. Alur, R. Singh, D. Fisman, and A. Solar-Lezama, "Search-based program synthesis", *Commun. ACM*, vol. 61, no. 12, pp. 84–93, Nov. 2018, ISSN: 0001-0782. [Online]. Available: https://doi.org/10.1145/3208071.

[5] "Backports project", https://backports.wiki.kernel.org/index.php/Main_Page, Accessed: 2020-12-20, 2020.

[6] J. Bader, A. Scott, M. Pradel, and S. Chandra, "Getafix: Learning to fix bugs automatically", *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, 159:1–159:27, Oct. 2019. [Online]. Available: https://doi.org/10.1145/3360585.

[7] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques", *ACM Comput. Surv.*, vol. 51, no. 3, 50:1–50:39, May 2018, ISSN: 0360-0300. [Online]. Available: http://doi.acm.org/10.1145/3182657.

[8] T. Bao, R. Wang, Y. Shoshitaishvili, and D. Brumley, "Your exploit is mine: Automatic shellcode transplant for remote exploits", in *2017 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2017, pp. 824–839.

[9] E. T. Barr, M. Harman, Y. Jia, A. Marginean, and J. Petke, “Automated software transplantation”, in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ACM, 2015, pp. 257–269.

[10] R. Bavishi, H. Yoshida, and M. R. Prasad, “Phoenix: Automated data-driven synthesis of repairs for static analysis violations”, in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE’19, New York, NY, USA: Association for Computing Machinery, 2019, pp. 613–624, ISBN: 9781450355728. [Online]. Available: `https://doi.org/10.1145/3338906.3338952`.

[11] I. Bertram, J. Hong, Y. Huang, W. Weimer, and Z. Sharafi, “Trustworthiness Perceptions in Code Review: An Eye-Tracking Study”, in *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, ser. ESEM ’20, New York, NY, USA: Association for Computing Machinery, 2020, ISBN: 9781450375801. [Online]. Available: `https://doi.org/10.1145/3382494.3422164`.

[12] B. W. Boehm, “Software engineering economics”, in *Pioneers and Their Contributions to Software Engineering: sd&m Conference on Software Pioneers, Bonn, June 28/29, 2001, Original Historic Contributions*, M. Broy and E. Denert, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 99–150, ISBN: 978-3-642-48354-7. [Online]. Available: `https://doi.org/10.1007/978-3-642-48354-7_5`.

[13] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, “Directed greybox fuzzing”, in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’17, Dallas, Texas, USA: Association for Computing Machinery, 2017, pp. 2329–2344, ISBN: 9781450349468. [Online]. Available: `https://doi.org/10.1145/3133956.3134020`.

[14] M. Böhme, V.-T. Pham, and A. Roychoudhury, “Coverage-based greybox fuzzing as markov chain”, in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’16, Vienna, Austria: Association for Computing Machinery, 2016, pp. 1032–1043, ISBN:

9781450341394. [Online]. Available: https://doi.org/10.1145/2976749.2978428.

[15] M. Botinčan, M. Parkinson, and W. Schulte, "Separation logic verification of c programs with an smt solver", *Electron. Notes Theor. Comput. Sci.*, vol. 254, pp. 5–23, Oct. 2009, ISSN: 1571-0661. [Online]. Available: http://dx.doi.org/10.1016/j.entcs.2009.09.057.

[16] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs", in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08, San Diego, California: USENIX Association, 2008, pp. 209–224. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855741.1855756.

[17] J. P. Cambronero, J. Shen, J. Cito, E. Glassman, and M. Rinard, "Characterizing developer use of automatically generated patches", in *2019 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2019, pp. 181–185.

[18] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code", in *2012 IEEE Symposium on Security and Privacy*, May 2012, pp. 380–394.

[19] S. Chakraborty, Y. Ding, M. Allamanis, and B. Ray, "Codit: Code editing with tree-based neural models", *IEEE Transactions on Software Engineering*, vol. 48, no. 4, pp. 1385–1399, 2022.

[20] S. Chakraborty and B. Ray, "On multi-modal learning of editing source code", in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2021, pp. 443–455.

[21] S. Chakraborty, K. S. Meel, and M. Y. Vardi, "A scalable approximate model counter", in *Principles and Practice of Constraint Programming - 19th International Conference, CP 2013, Uppsala, Sweden, September 16-20, 2013. Proceedings*, C. Schulte, Ed., ser. Lecture Notes in Computer Science, vol. 8124, Springer, 2013, pp. 200–216. [Online]. Available: https://doi.org/10.1007/978-3-642-40627-0%5C_18.

[22] S. Chandra, S. J. Fink, and M. Sridharan, "Snugglebug: A powerful approach to weakest preconditions", in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '09, Dublin, Ireland: ACM, 2009, pp. 363–374, ISBN: 978-1-60558-392-1. [Online]. Available: `http://doi.acm.org/10.1145/1542476.1542517`.

[23] L. Chen, Y. Ouyang, and L. Zhang, "Fast and precise on-the-fly patch validation for all", in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 1123–1134.

[24] Q. Chen and R. A. Bridges, "Automated behavioral analysis of malware: A case study of wannacry ransomware", in *2017 16th IEEE International Conference on Machine Learning and Applications (ICMLA)*, 2017, pp. 454–460.

[25] Z. Chen, S. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus, "Sequencer: Sequence-to-sequence learning for end-to-end program repair", *IEEE Transactions on Software Engineering*, vol. 47, no. 9, pp. 1943–1959, 2021.

[26] V. Chipounov, V. Kuznetsov, and G. Candea, "S2e: A platform for in-vivo multi-path analysis of software systems", *SIGPLAN Not.*, vol. 47, no. 4, pp. 265–278, Mar. 2011, ISSN: 0362-1340. [Online]. Available: `http://doi.acm.org/10.1145/2248487.1950396`.

[27] I. Chowdhury and M. Zulkernine, "Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities", *Journal of Systems Architecture*, vol. 57, no. 3, pp. 294–313, 2011, ISSN: 1383–7621.

[28] A. Decan, T. Mens, A. Zerouali, and C. De Roover, "Back to the past – analysing backporting practices in package dependency networks", *IEEE Transactions on Software Engineering*, pp. 1–1, 2021.

[29] P. Dinges and G. Agha, "Targeted test input generation using symbolic-concrete backward execution", in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '14, Vasteras, Sweden: ACM, 2014, pp. 31–36, ISBN: 978-1-4503-3013-8. [Online]. Available: `http://doi.acm.org/10.1145/2642937.2642951`.

[30] G. J. Duck, X. Gao, and A. Roychoudhury, “Binary rewriting without control flow recovery”,, ser. PLDI 2020, London, UK: Association for Computing Machinery, 2020, pp. 151–163, ISBN: 9781450376136. [Online]. Available: https://doi.org/10.1145/3385412.3385972.

[31] T. Durieux, B. Cornu, L. Seinturier, and M. Monperrus, “Dynamic patch generation for null pointer exceptions using metaprogramming”, in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2017, pp. 349–358.

[32] Z. Durumeric, F. Li, J. Kasten, J. Amann, J. Beekman, M. Payer, N. Weaver, D. Adrian, V. Paxson, M. Bailey, *et al.*, “The matter of heartbleed”, in *Proceedings of the 2014 conference on internet measurement conference*, ACM, 2014, pp. 475–488.

[33] K. Ehrich, “Mechanical turk: Potential concerns and their solutions”, https://www.summitllc.us/blog/mechanical-turk-concerns-and-solutions, 2020.

[34] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, “Fine-grained and accurate source code differencing”, in *ACM/IEEE International Conference on Automated Software Engineering, ASE ’14, Vasteras, Sweden - September 15 - 19, 2014*, New York, NY, USA: Association for Computing Machinery, 2014, pp. 313–324, ISBN: 9781450330138. [Online]. Available: http://doi.acm.org/10.1145/2642937.2642982.

[35] B. Fluri, M. Wuersch, M. PInzger, and H. Gall, “Change distilling: Tree differencing for fine-grained source code change extraction”, *IEEE Transactions on software engineering*, vol. 33, no. 11, pp. 725–743, Nov. 2007, ISSN: 0098-5589. [Online]. Available: https://doi.org/10.1109/TSE.2007.70731.

[36] H. Frenkel, O. Grumberg, C. Pasareanu, and S. Sheinvald, “Assume, guarantee or repair”, in *Tools and Algorithms for the Construction and Analysis of Systems*, A. Biere and D. Parker, Eds., Cham: Springer International Publishing, 2020, pp. 211–227, ISBN: 978-3-030-45190-5.

[37] Z. P. Fry, B. Landau, and W. Weimer, “A human study of patch maintainability”, in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ser. ISSTA 2012, Minneapolis, MN, USA: ACM, 2012, pp. 177–187, ISBN: 978-1-4503-1454-1. [Online]. Available: `http://doi.acm.org/10.1145/2338965.2336775`.

[38] X. Gao, S. Barke, A. Radhakrishna, G. Soares, S. Gulwani, A. Leung, N. Nagappan, and A. Tiwari, “Feedback-driven semi-supervised synthesis of program transformations”, *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–30, 2020. [Online]. Available: `https://doi.org/10.1145/3428287`.

[39] X. Gao, S. Mechtaev, and A. Roychoudhury, “Crash-avoiding program repair”, in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, ser. ISSTA 2019, Beijing, China: Association for Computing Machinery, 2019, pp. 8–18, ISBN: 9781450362245. [Online]. Available: `https://doi.org/10.1145/3293882.3330558`.

[40] X. Gao, B. Wang, G. J. Duck, R. Ji, Y. Xiong, and A. Roychoudhury, “Beyond tests: Program vulnerability repair via crash constraint extraction”, *ACM Trans. Softw. Eng. Methodol.*, vol. 30, no. 2, Feb. 2021, ISSN: 1049-331X. [Online]. Available: `https://doi.org/10.1145/3418461`.

[41] A. Ghanbari, S. Benton, and L. Zhang, “Practical program repair via bytecode mutation”, in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2019, Beijing, China: Association for Computing Machinery, 2019, pp. 19–30, ISBN: 9781450362245. [Online]. Available: `https://doi.org/10.1145/3293882.3330559`.

[42] P. Godefroid, N. Klarlund, and K. Sen, “Dart: Directed automated random testing”, *SIGPLAN Not.*, vol. 40, no. 6, pp. 213–223, Jun. 2005, ISSN: 0362-1340. [Online]. Available: `http://doi.acm.org/10.1145/1064978.1065036`.

[43] P. Godefroid, M. Y. Levin, and D. Molnar, “Sage: Whitebox fuzzing for security testing”, *Queue*, vol. 10, no. 1, 20:20–20:27, Jan. 2012, ISSN: 1542-7730. [Online]. Available: `http://doi.acm.org/10.1145/2090147.2094081`.

[44] C. P. Gomes, A. Sabharwal, and B. Selman, “Model counting”, in *Handbook of Satisfiability*, ser. Frontiers in Artificial Intelligence and Applications, A. Biere, M. Heule, H. van Maaren, and T. Walsh, Eds., vol. 185, IOS Press, 2009, pp. 633–654. [Online]. Available: https://doi.org/10.3233/978-1-58603-929-5-633.

[45] D. Gopinath, M. Z. Malik, and S. Khurshid, “Specification-based program repair using sat”, in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, P. A. Abdulla and K. R. M. Leino, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 173–188, ISBN: 978-3-642-19835-9.

[46] C. L. Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. T. Devanbu, S. Forrest, and W. Weimer, “The manybugs and introclass benchmarks for automated repair of C programs”, *IEEE Trans. Software Eng.*, vol. 41, no. 12, pp. 1236–1256, 2015. [Online]. Available: https://doi.org/10.1109/TSE.2015.2454513.

[47] “Guidelines for backporting linux patches”, https://www.kernel.org/doc/Documentation/process/stable-kernel-rules.rst, Accessed: 2021-01-24, 2021.

[48] R. Gupta, S. Pal, A. Kanade, and S. Shevade, “Deepfix: Fixing common c language errors by deep learning”, *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 31, no. 1, Feb. 2017. [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/10742.

[49] R. Hiesgen, M. Nawrocki, T. C. Schmidt, and M. Wählisch, “The race to the vulnerable: Measuring the log4j shell incident”, 2022. [Online]. Available: https://arxiv.org/abs/2205.02544.

[50] W. E. Howden, “Symbolic testing and the dissect symbolic evaluation system”, *IEEE Transactions on Software Engineering*, vol. SE-3, no. 4, pp. 266–278, Jul. 1977, ISSN: 0098-5589.

[51] Q. Hu, J. Cyphert, L. D’Antoni, and T. Reps, “Exact and approximate methods for proving unrealizability of syntax-guided synthesis problems”,, ser. PLDI 2020, London, UK: Association for Computing Machinery, 2020,

pp. 1128–1142, ISBN: 9781450376136. [Online]. Available: `https://doi.org/10.1145/3385412.3385979`.

[52] Z. Huang, D. Lie, G. Tan, and T. Jaeger, "Using safety properties to generate vulnerability patches", in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 539–554.

[53] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari, "Oracle-guided component-based program synthesis",, ser. ICSE '10, Cape Town, South Africa: Association for Computing Machinery, 2010, pp. 215–224, ISBN: 9781605587196. [Online]. Available: `https://doi.org/10.1145/1806799.1806833`.

[54] J. Jiang, L. Ren, Y. Xiong, and L. Zhang, "Inferring program transformations from singular examples via big code", in *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*, ser. ASE '19, San Diego, California: IEEE, 2019, pp. 255–266, ISBN: 9781728125084. [Online]. Available: `https://doi.org/10.1109/ASE.2019.00033`.

[55] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones", in *Proceedings of the 29th International Conference on Software Engineering*, ser. ICSE '07, USA: IEEE Computer Society, 2007, pp. 96–105, ISBN: 0769528287. [Online]. Available: `https://doi.org/10.1109/ICSE.2007.30`.

[56] L. Jiang, Z. Su, and E. Chiu, "Context-based detection of clone-related bugs", in *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC-FSE '07, Dubrovnik, Croatia: ACM, 2007, pp. 55–64, ISBN: 978-1-59593-811-4. [Online]. Available: `http://doi.acm.org/10.1145/1287624.1287634`.

[57] N. Jiang, T. Lutellier, and L. Tan, "Cure: Code-aware neural machine translation for automatic program repair", in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 1161–1173.

[58] R. Just, M. D. Ernst, and G. Fraser, “Efficient mutation analysis by propagating and partitioning infected execution states”, in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014, San Jose, CA, USA: Association for Computing Machinery, 2014, pp. 315–326, ISBN: 9781450326452. [Online]. Available: `https://doi.org/10.1145/2610384.2610388`.

[59] D. Kim, J. Nam, J. Song, and S. Kim, “Automatic patch generation learned from human-written patches”, in *ICSE' 2013*, ser. ICSE '13, IEEE Press, San Francisco, CA, USA: IEEE Press, 2013, pp. 802–811, ISBN: 9781467330763.

[60] J. C. King, “A new approach to program testing”, in *Proceedings of the International Conference on Reliable Software*, Los Angeles, California: ACM, 1975, pp. 228–233. [Online]. Available: `http://doi.acm.org/10.1145/800027.808444`.

[61] J. C. King, “Symbolic execution and program testing”, *Commun. ACM*, vol. 19, no. 7, pp. 385–394, Jul. 1976, ISSN: 0001-0782. [Online]. Available: `http://doi.acm.org/10.1145/360248.360252`.

[62] S. Kirbas, E. Windels, O. McBello, K. Kells, M. Pagano, R. Szalanski, V. Nowack, E. Winter, S. Counsell, D. Bowes, T. Hall, S. Haraldsson, and J. Woodward, “On the introduction of automatic program repair in bloomberg”, *IEEE Software*, vol. 38, no. 04, pp. 43–51, Jul. 2021, ISSN: 1937-4194.

[63] X. Kong, L. Zhang, W. E. Wong, and B. Li, “The impacts of techniques, programs and tests on automated program repair: An empirical study”, *Journal of Systems and Software*, vol. 137, pp. 480–496, 2018, ISSN: 0164-1212. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/S0164121217301279`.

[64] R. Könighofer and R. Bloem, “Repair with on-the-fly program analysis”, in *Hardware and Software: Verification and Testing*, A. Biere, A. Nahir, and T. Vos, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 56–71, ISBN: 978-3-642-39611-3.

[65] J. Lawall and G. Muller, "Coccinelle: 10 years of automated evolution in the linux kernel", in *USENIX Annual Technical Conference*, ser. USENIX ATC '18, USA: USENIX Association, 2018, pp. 601–614, ISBN: 9781931971447. [Online]. Available: `https://hal.inria.fr/hal-01853271`.

[66] X.-B. D. Le, D.-H. Chu, D. Lo, C. Le Goues, and W. Visser, "S3: Syntax- and semantic-guided repair synthesis via programming by examples", in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017, Paderborn, Germany: Association for Computing Machinery, 2017, pp. 593–604, ISBN: 9781450351058. [Online]. Available: `https://doi.org/10.1145/3106237.3106309`.

[67] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each", in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12, Zurich, Switzerland: IEEE Press, 2012, pp. 3–13, ISBN: 978-1-4673-1067-3. [Online]. Available: `http://dl.acm.org/citation.cfm?id=2337223.2337225`.

[68] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair", *IEEE Trans. Softw. Eng.*, vol. 38, no. 1, pp. 54–72, Jan. 2012, ISSN: 0098-5589. [Online]. Available: `http://dx.doi.org/10.1109/TSE.2011.104`.

[69] C. Le Goues, M. Pradel, and A. Roychoudhury, "Automated program repair", *Communications of the ACM*, vol. 62, no. 12, pp. 56–65, 12 Nov. 2019, ISSN: 0001-0782. [Online]. Available: `https://doi.org/10.1145/3318162`.

[70] J. D. Lee and K. A. See, "Trust in automation: Designing for appropriate reliance", *Human Factors*, vol. 46, no. 1, pp. 50–80, 2004. eprint: `https://doi.org/10.1518/hfes.46.1.50\_30392`. [Online]. Available: `https://doi.org/10.1518/hfes.46.1.50%5C_30392`.

[71] J. Liang, R. Ji, J. Jiang, Y. Lou, Y. Xiong, and G. Huang, "Interactive patch filtering as debugging aid", *arXiv preprint arXiv:2004.08746*, 2020.

[72] "Linux patch tool", `https://man7.org/linux/man-pages/man1/patch.1.html`, Accessed: 2020-11-20, 2021.

[73] K. Liu, L. Li, A. Koyuncu, D. Kim, Z. Liu, J. Klein, and T. F. Bissyandé, "A critical review on the evaluation of automated program repair systems", *Journal of Systems and Software*, vol. 171, p. 110 817, 2021, ISSN: 0164-1212. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121220302156.

[74] K. Liu, S. Wang, A. Koyuncu, K. Kim, T. F. Bissyandé, D. Kim, P. Wu, J. Klein, X. Mao, and Y. L. Traon, "On the efficiency of test suite based program repair: A systematic assessment of 16 automated repair systems for java programs", in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20, Seoul, South Korea: Association for Computing Machinery, 2020, pp. 615–627, ISBN: 9781450371216. [Online]. Available: https://doi.org/10.1145/3377811.3380338.

[75] "Llvm gumtree implementation", https://github.com/llvm/llvm-project/main/clang/tools/clang-diff, Accessed: 2021-1-30, 2021.

[76] F. Logozzo and T. Ball, "Modular and verified automatic program repair", in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '12, Tucson, Arizona, USA: Association for Computing Machinery, 2012, pp. 133–146, ISBN: 9781450315616. [Online]. Available: https://doi.org/10.1145/2384616.2384626.

[77] F. Long, P. Amidon, and M. Rinard, "Automatic inference of code transforms for patch generation", in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017, Paderborn, Germany: Association for Computing Machinery, 2017, pp. 727–739, ISBN: 9781450351058. [Online]. Available: https://doi.org/10.1145/3106237.3106253.

[78] F. Long and M. Rinard, "Staged program repair with condition synthesis", in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015, Bergamo, Italy: Association for Computing Machinery, 2015, pp. 166–178, ISBN: 9781450336758. [Online]. Available: https://doi.org/10.1145/2786805.2786811.

[79] F. Long and M. Rinard, “Automatic patch generation by learning correct code”, in *43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL ’16, St. Petersburg, FL, USA: ACM, 2016, pp. 298–312, ISBN: 978-1-4503-3549-2. [Online]. Available: https://doi.org/10.1145/2837614.2837617.

[80] F. Long and M. C. Rinard, “An analysis of the search spaces for generate and validate patch generation systems”, in *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, L. K. Dillon, W. Visser, and L. A. Williams, Eds., ACM, 2016, pp. 702–713. [Online]. Available: https://doi.org/10.1145/2884781.2884872.

[81] Y. Lou, S. Benton, D. Hao, L. Zhang, and L. Zhang, “How does regression test selection affect program repair? an extensive study on 2 million patches”, *arXiv preprint arXiv:2105.07311*, 2021.

[82] Y. Lu, S. Chaudhuri, C. Jermaine, and D. Melski, “Program splicing”, in *Proceedings of the 40th International Conference on Software Engineering*, ACM, 2018, pp. 338–349.

[83] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan, “Coconut: Combining context-aware neural translation models using ensemble for program repair”, in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2020, Virtual Event, USA: Association for Computing Machinery, 2020, pp. 101–114, ISBN: 9781450380089. [Online]. Available: https://doi.org/10.1145/3395363.3397369.

[84] K.-K. Ma, K. Y. Phang, J. S. Foster, and M. Hicks, “Directed symbolic execution”, in *Proceedings of the 18th International Conference on Static Analysis*, ser. SAS’11, Venice, Italy: Springer-Verlag, 2011, pp. 95–111, ISBN: 978-3-642-23701-0. [Online]. Available: http://dl.acm.org/citation.cfm?id=2041552.2041563.

[85] A. Marginean, J. Bader, S. Chandra, M. Harman, Y. Jia, K. Mao, A. Mols, and A. Scott, “Sapfix: Automated end-to-end repair at scale”, in *2019 IEEE/ACM*

41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), 2019, pp. 269–278.

[86] A. Marginean, E. T. Barr, M. Harman, and Y. Jia, “Automated transplantation of call graph and layout features into kate”, in *International Symposium on Search Based Software Engineering*, Springer, 2015, pp. 262–268.

[87] M. Martinez, T. Durieux, R. Sommerard, J. Xuan, and M. Monperrus, “Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset”, *Empirical Softw. Engg.*, vol. 22, no. 4, pp. 1936–1964, Aug. 2017, ISSN: 1382-3256. [Online]. Available: `https://doi.org/10.1007/s10664-016-9470-4`.

[88] M. Martinez and M. Monperrus, “Astor: A program repair library for java (demo)”, in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016, Saarbr&#252;cken, Germany: ACM, 2016, pp. 441–444, ISBN: 978-1-4503-4390-9. [Online]. Available: `http://doi.acm.org/10.1145/2931037.2948705`.

[89] S. Mechtaev, X. Gao, S. H. Tan, and A. Roychoudhury, “Test-equivalence analysis for automatic patch generation”, *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 27, no. 4, 4 Oct. 2018, ISSN: 1049-331X. [Online]. Available: `https://doi.org/10.1145/3241980`.

[90] S. Mechtaev, A. Griggio, A. Cimatti, and A. Roychoudhury, “Symbolic execution with existential second-order constraints”, in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018, Lake Buena Vista, FL, USA: Association for Computing Machinery, 2018, pp. 389–399, ISBN: 9781450355735. [Online]. Available: `https://doi.org/10.1145/3236024.3236049`.

[91] S. Mechtaev, J. Yi, and A. Roychoudhury, “Directfix: Looking for simple program repairs”, in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, IEEE Press, 2015, pp. 448–458.

[92] S. Mechtaev, J. Yi, and A. Roychoudhury, “Angelix: Scalable multiline program patch synthesis via symbolic analysis”, in *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*, ser. ICSE ’16, IEEE, Austin, Texas: Association for Computing Machinery, 2016, pp. 691–701, ISBN: 9781450339001. [Online]. Available: `https://doi.org/10.1145/2884781.2884807`.

[93] N. Meng, M. Kim, and K. S. McKinley, “Lase: Locating and applying systematic edits by learning from examples”, in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE ’13, IEEE Press, San Francisco, CA, USA: IEEE Press, 2013, pp. 502–511, ISBN: 978-1-4673-3076-3. [Online]. Available: `http://dl.acm.org/citation.cfm?id=2486788.2486855`.

[94] N. Meng, M. Kim, and K. S. McKinley, “Systematic editing: Generating program transformations from an example”, in *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’11, San Jose, California, USA: ACM, 2011, pp. 329–342, ISBN: 978-1-4503-0663-8. [Online]. Available: `http://doi.acm.org/10.1145/1993498.1993537`.

[95] A. Miltner, S. Gulwani, V. Le, A. Leung, A. Radhakrishna, G. Soares, A. Tiwari, and A. Udupa, “On the fly synthesis of edit suggestions”, *Proceedings of the ACM on Programming Languages*, vol. 3, pp. 1–29, Oct. 2019. [Online]. Available: `https://doi.org/10.1145/3360569`.

[96] T. Molderez, R. Stevens, and C. De Roover, “Mining change histories for unknown systematic edits”, in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories*, ser. MSR ’17, IEEE, Buenos Aires, Argentina, 2017, pp. 248–256. [Online]. Available: `https://doi.org/10.1109/MSR.2017.12`.

[97] M. Monperrus, “Automatic software repair: A bibliography”, *ACM Comput. Surv.*, vol. 51, no. 1, Jan. 2018, ISSN: 0360-0300. [Online]. Available: `https://doi.org/10.1145/3105906`.

[98] M. Motwani, M. Soto, Y. Brun, R. Just, and C. Le Goues, "Quality of Automated Program Repair on Real-World Defects", *IEEE Transactions on Software Engineering*, p. 1, 2020.

[99] H. A. Nguyen, T. N. Nguyen, D. Dig, S. Nguyen, H. Tran, and M. Hilton, "Graph-based mining of in-the-wild, fine-grained, semantic code change patterns", in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE '19, IEEE, Montreal, Quebec, Canada, 2019, pp. 819–830. [Online]. Available: `https://doi.org/10.1109/ICSE.2019.00089`.

[100] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: Program repair via semantic analysis", in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13, San Francisco, CA, USA: IEEE Press, 2013, pp. 772–781, ISBN: 978-1-4673-3076-3. [Online]. Available: `http://dl.acm.org/citation.cfm?id=2486788.2486890`.

[101] Y. Noller, R. Shariffdeen, X. Gao, and A. Roychoudhury, "Trust enhancement issues in program repair",, 2022. [Online]. Available: `https://doi.org/10.1145/3510003.3510040`.

[102] Y. Padioleau, J. Lawall, R. Hansen, and G. Muller, "Documenting and automating collateral evolutions in linux device drivers", in *ACM SIGOPS/EuroSys European Conference on Computer Systems (Eurosys)*, ser. EDCC'15, USA: IEEE Computer Society, 2008, pp. 132–143, ISBN: 9781467392891. [Online]. Available: `https://doi.org/10.1109/EDCC.2015.23`.

[103] N. Palix, G. Thomas, S. Saha, C. Calvès, J. Lawall, and G. Muller, "Faults in linux: Ten years later", in *ASPLOS*, 2011.

[104] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?" In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ser. ISSTA '11, Toronto, Ontario, Canada: Association for Computing Machinery, 2011, pp. 199–209, ISBN: 9781450305624. [Online]. Available: `https://doi.org/10.1145/2001420.2001445`.

[105] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. Rinard, "Automatically patching errors in deployed

software”, in *SOSP*, ser. SOSP ’09, Big Sky, Montana, USA: Association for Computing Machinery, 2009, pp. 87–102, ISBN: 9781605587523. [Online]. Available: https://doi.org/10.1145/1629575.1629585.

[106] N. H. Pham, T. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, “Detection of recurring software vulnerabilities”, in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE ’10, Antwerp, Belgium: ACM, 2010, pp. 447–456, ISBN: 978-1-4503-0116-9. [Online]. Available: http://doi.acm.org/10.1145/1858996.1859089.

[107] V.-T. Pham, M. Böhme, and A. Roychoudhury, “Model-based whitebox fuzzing for program binaries”, in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2016, Singapore, Singapore: Association for Computing Machinery, 2016, pp. 543–553, ISBN: 9781450338455. [Online]. Available: https://doi.org/10.1145/2970276.2970316.

[108] V.-T. Pham, M. Böhme, A. E. Santosa, A. R. Căciulescu, and A. Roychoudhury, “Smart greybox fuzzing”, *IEEE Transactions on Software Engineering*, vol. 47, no. 9, pp. 1980–1997, 2021.

[109] G. D. Plotkin, “A note on inductive generalization”, *Machine intelligence*, vol. 5, no. 1, pp. 153–163, 1970.

[110] Z. Qi, F. Long, S. Achour, and M. Rinard, “An analysis of patch plausibility and correctness for generate-and-validate patch generation systems”, in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ser. ISSTA 2015, Baltimore, MD, USA: ACM, 2015, pp. 24–36, ISBN: 978-1-4503-3620-8. [Online]. Available: http://doi.acm.org/10.1145/2771783.2771791.

[111] B. Ray and M. Kim, “A case study of cross-system porting in forked projects”, in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE ’12, Cary, North Carolina: ACM, 2012, 53:1–53:11, ISBN: 978-1-4503-1614-9.

[112] B. Ray, M. Kim, S. Person, and N. Rungta, "Detecting and characterizing semantic inconsistencies in ported code", in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ser. ASE'13, IEEE, Silicon Valley, CA, USA: IEEE Press, 2013, pp. 367–377, ISBN: 978-1-4799-0215-6. [Online]. Available: `https://doi.org/10.1109/ASE.2013.6693095`.

[113] R. Rolim, G. Soares, L. D'Antoni, O. Polozov, S. Gulwani, R. Gheyi, R. Suzuki, and B. Hartmann, "Learning syntactic program transformations from examples", in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE '17, IEEE Press, Buenos Aires, Argentina: IEEE Press, 2017, pp. 404–415, ISBN: 9781538638682. [Online]. Available: `https://doi.org/10.1109/ICSE.2017.44`.

[114] C. K. Roy and J. R. Cordy, "An empirical study of function clones in open source software", in *2008 15th Working Conference on Reverse Engineering*, ser. WCRE '08, Washington, DC, USA: IEEE Computer Society, Oct. 2008, pp. 81–90, ISBN: 978-0-7695-3429-9. [Online]. Available: `https://doi.org/10.1109/WCRE.2008.54`.

[115] T. J. Ryan, G. M. Alarcon, C. Walter, R. Gamble, S. A. Jessup, A. Capiola, and M. D. Pfahler, "Trust in automated software repair", in *HCI for Cybersecurity, Privacy and Trust*, A. Moallem, Ed., Cham: Springer International Publishing, 2019, pp. 452–470, ISBN: 978-3-030-22351-9.

[116] G. Sakkas, M. Endres, B. Cosman, W. Weimer, and R. Jhala, "Type error feedback via analytic program repair", in *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, A. F. Donaldson and E. Torlak, Eds., ACM, 2020, pp. 16–30. [Online]. Available: `https://doi.org/10.1145/3385412.3386005`.

[117] M. Schreier, "Qualitative content analysis in practice", Sage publications, 2012.

[118] W. H. Security, "AppSec Stats Flash: 2021 Year in Review", https://www.whitehatsec.com/blog/appsec-stats-flash-2021-year-in-review/, [Online; accessed 24-May-2022], 2022.

[119] K. Sen, "Concolic testing", in *Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '07, Atlanta, Georgia, USA: Association for Computing Machinery, 2007, pp. 571–572, ISBN: 9781595938824. [Online]. Available: https://doi.org/10.1145/1321631.1321746.

[120] L. Serrano, V.-A. Nguyen, F. Thung, L. Jiang, D. Lo, J. Lawall, and G. Muller, "SPINFER: inferring semantic patches for the Linux kernel", in *USENIX Annual Technical Conference*, ser. USENIX ATC'20, USENIX Association, Jul. 2020, pp. 235–248. [Online]. Available: https://www.usenix.org/conference/atc20/presentation/serrano.

[121] M. Shahzad, M. Z. Shafiq, and A. X. Liu, "A large scale exploratory analysis of software vulnerability life cycles", in *Software Engineering (ICSE), 2012 34th International Conference on*, IEEE, 2012, pp. 771–781.

[122] S. Shen, A. Kolluri, Z. Dong, P. Saxena, and A. Roychoudhury, "Localizing vulnerabilities statistically from one exploit", in *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, ser. ASIA CCS '21, Virtual Event, Hong Kong: Association for Computing Machinery, 2021, pp. 537–549, ISBN: 9781450382878. [Online]. Available: https://doi.org/10.1145/3433210.3437528.

[123] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne, "Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities", *IEEE Transactions on Software Engineering*, vol. 37, no. 6, pp. 772–787, Nov. 2011, ISSN: 0098-5589.

[124] Shodan, "Devices Vulnerable to Heartbleed", https://www.shodan.io/report/89bnfUyJ, [Online; accessed 14-September-2018], 2016.

[125] S. Sidiroglou-Douskos, E. Lahtinen, A. Eden, F. Long, and M. Rinard, "Codecarboncopy", in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017, ACM, Paderborn, Germany:

Association for Computing Machinery, 2017, pp. 95–105, ISBN: 9781450351058. [Online]. Available: https://doi.org/10.1145/3106237.3106269.

[126] S. Sidiroglou-Douskos, E. Lahtinen, F. Long, and M. Rinard, "Automatic error elimination by horizontal code transfer across multiple applications", in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '15, vol. 50, Portland, OR, USA: ACM, Jun. 2015, pp. 43–54, ISBN: 978-1-4503-3468-6. [Online]. Available: http://doi.acm.org/10.1145/2737924.2737988.

[127] S. Sidiroglou-Douskos, E. Lahtinen, and M. Rinard, "Automatic discovery and patching of buffer and integer overflow errors", Massachusetts Institute of Technology, Cambridge, MA, USA, Tech. Rep., 2015. [Online]. Available: http://hdl.handle.net/1721.1/97087.

[128] "Site for heartbleed bug", http://heartbleed.com/, Accessed: 2019-03-13, 2019.

[129] "Site for krack patches", https://github.com/kristate/krackinfo/, Accessed: 2019-03-13.

[130] "Site for krack vulnerability", https://www.krackattacks.com/, Accessed: 2019-03-13, 2019.

[131] "Site for shellshock bug", https://shellshock.io/, Accessed: 2019-03-13, 2019.

[132] "Site for zipslip bug", https://snyk.io/research/zip-slip-vulnerability, 2019.

[133] E. K. Smith, E. T. Barr, C. Le Goues, and Y. Brun, "Is the cure worse than the disease? overfitting in automated program repair", in *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015, Bergamo, Italy: ACM, 2015, pp. 532–543, ISBN: 978-1-4503-3675-8. [Online]. Available: http://doi.acm.org/10.1145/2786805.2786825.

[134] A. Solar-Lezama, "Program synthesis by sketching", Ph.D. dissertation, EECS Department, University of California, Berkeley, 2008.

[135] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat, “Combinatorial sketching for finite programs”, in *International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS*, 2006, ISBN: 1595934510.

[136] S. H. Tan, Z. Dong, X. Gao, and A. Roychoudhury, “Repairing crashes in android apps”, in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE ’18, IEEE, Gothenburg, Sweden: ACM, 2018, pp. 187–198, ISBN: 978-1-4503-5638-1. [Online]. Available: `http://doi.acm.org/10.1145/3180155.3180243`.

[137] S. H. Tan and A. Roychoudhury, “Relifix: Automated repair of software regressions”, in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE ’15, IEEE, vol. 1, Florence, Italy: IEEE Press, 2015, pp. 471–482, ISBN: 978-1-4799-1934-5. [Online]. Available: `http://dl.acm.org/citation.cfm?id=2818754.2818813`.

[138] S. H. Tan, H. Yoshida, M. R. Prasad, and A. Roychoudhury, “Anti-patterns in search-based program repair”, in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016, ACM, Seattle, WA, USA: Association for Computing Machinery, 2016, pp. 727–738, ISBN: 9781450342186. [Online]. Available: `https://doi.org/10.1145/2950290.2950295`.

[139] Y. Tao, J. Kim, S. Kim, and C. Xu, “Automatically generated patches as debugging aids: A human study”, in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014, Hong Kong, China: Association for Computing Machinery, 2014, pp. 64–74, ISBN: 9781450330565. [Online]. Available: `https://doi.org/10.1145/2635868.2635873`.

[140] “The clang project”, `https://clang.llvm.org`, Accessed: 2019-03-11, 2019.

[141] “The docker project”, `https://www.docker.com/`, Accessed: 2020-03-05, 2020.

[142] “The z3 theorem prover”, `https://github.com/Z3Prover/z3`, Accessed: 2019-03-11, 2019.

[143] F. Thung, X.-B. D. Le, D. Lo, and J. L. Lawall, "Recommending code changes for automatic backporting of linux device drivers", in *2016 IEEE International Conference on Software Maintenance and Evolution, ICSME 2016, Raleigh, NC, USA, October 2-7, 2016*, ser. ICSME'16, IEEE, IEEE Computer Society, 2016, pp. 222–232. [Online]. Available: https://doi.org/10.1109/ICSME.2016.71.

[144] Y. Tian, J. Lawall, and D. Lo, "Identifying linux bug fixing patches", in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12, IEEE, Zurich, Switzerland: IEEE Press, 2012, pp. 386–396, ISBN: 978-1-4673-1067-3. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337223.2337269.

[145] C. Timperley *et al.*, "Darjeeling: Language agnostic search-based repair tool", https://github.com/squaresLab/Darjeeling.

[146] R. van Tonder and C. Le Goues, "Static automated program repair for heap properties", in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, IEEE, 2018, pp. 151–162.

[147] M. Tufano, J. Pantiuchina, C. Watson, G. Bavota, and D. Poshyvanyk, "On learning meaningful code changes via neural machine translation", in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE '19, Montreal, Quebec, Canada: IEEE Press, 2019, pp. 25–36. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00021.

[148] S. Urli, Z. Yu, L. Seinturier, and M. Monperrus, "How to design a program repair bot? insights from the repairnator project", in *2018 IEEE/ACM 40th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, 2018, pp. 95–104.

[149] "Us-cert bulletins", https://us-cert.cisa.gov/.

[150] K. S. Wan, "Notpetya, not warfare: Rethinking the insurance war exclusion in the context of international cyberattacks", *Wash. L. Rev.*, vol. 95, p. 1595, 2020.

[151] S. Wang, M. Wen, L. Chen, X. Yi, and X. Mao, “How Different Is It Between Machine-Generated and Developer-Provided Patches? : An Empirical Study on the Correct Patches Generated by Automated Program Repair Techniques”, in *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2019, pp. 1–12.

[152] S.-C. Website, “International competition on software verification (sv-comp)”, `https://sv-comp.sosy-lab.org/`, 2020.

[153] W. Weimer, S. Forrest, M. Kim, C. Le Goues, and P. Hurley, “Trusted Software Repair for System Resiliency”, in *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshop (DSN-W)*, 2016, pp. 238–241.

[154] W. Weimer, Z. P. Fry, and S. Forrest, “Leveraging program equivalence for adaptive program repair: Models and first results”, in *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE’13, Silicon Valley, CA, USA: IEEE Press, 2013, pp. 356–366, ISBN: 978-1-4799-0215-6. [Online]. Available: `https://doi.org/10.1109/ASE.2013.6693094`.

[155] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, “Automatically finding patches using genetic programming”, in *ICSE*, 2009, pp. 364–374.

[156] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung, “An empirical analysis of the influence of fault space on search-based automated program repair”, *arXiv preprint arXiv:1707.05172*, 2017.

[157] C.-P. Wong, P. Santiesteban, C. Kästner, and C. Le Goues, “Varfix: Balancing edit expressiveness and search effectiveness in automated program repair”, in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2021, Athens, Greece: Association for Computing Machinery, 2021, pp. 354–366, ISBN: 9781450385626. [Online]. Available: `https://doi.org/10.1145/3468264.3468600`.

[158] Q. Xin and S. P. Reiss, "Identifying test-suite-overfitted patches through test case generation", in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2017, Santa Barbara, CA, USA: ACM, 2017, pp. 226–236, ISBN: 978-1-4503-5076-1. [Online]. Available: `http://doi.acm.org/10.1145/3092703.3092718`.

[159] Y. Xiong, X. Liu, M. Zeng, L. Zhang, and G. Huang, "Identifying patch correctness in test-based program repair", in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18, Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 789–799, ISBN: 9781450356381. [Online]. Available: `https://doi.org/10.1145/3180155.3180182`.

[160] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang, "Precise condition synthesis for program repair", in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE '17, Buenos Aires, Argentina: IEEE Press, 2017, pp. 416–426, ISBN: 978-1-5386-3868-2. [Online]. Available: `https://doi.org/10.1109/ICSE.2017.45`.

[161] J. Xuan, M. Martinez, F. DeMarco, M. Clement, S. L. Marcote, T. Durieux, D. L. Berre, and M. Monperrus, "Nopol: Automatic repair of conditional statement bugs in java programs", *IEEE Transactions on Software Engineering*, vol. PP, no. 99, pp. 1–1, 2016, ISSN: 0098-5589.

[162] F. Yamaguchi, C. Wressnegger, H. Gascon, and K. Rieck, "Chucky: Exposing missing checks in source code for vulnerability discovery", in *ACM Conference on Computer and Communications Security*, 2013.

[163] D. Yang, Y. Qi, X. Mao, and Y. Lei, "Evaluating the usage of fault localization in automated program repair: an empirical study", *Frontiers of Computer Science*, vol. 15, no. 1, p. 151 202, 2020, ISSN: 2095-2236. [Online]. Available: `https://doi.org/10.1007/s11704-020-9263-1`.

[164] J. Yang, A. Zhikhartsev, Y. Liu, and L. Tan, "Better test cases for better automated program repair", in *Joint Meeting on Foundations of Software Engineering (ESEC-FSE)*, 2017.

[165] H. Ye, M. Martinez, and M. Monperrus, “Neural program repair with execution-based backpropagation”,, 2022. [Online]. Available: `https://doi.org/10.1145/3510003.3510222`.

[166] T. Zhang and M. Kim, “Automated transplantation and differential testing for clones”, in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE ’17, IEEE Press, Buenos Aires, Argentina: IEEE Press, 2017, pp. 665–676, ISBN: 978-1-5386-3868-2. [Online]. Available: `https://doi.org/10.1109/ICSE.2017.67`.

[167] Q. Zhu, Z. Sun, Y.-a. Xiao, W. Zhang, K. Yuan, Y. Xiong, and L. Zhang, “A syntax-guided edit decoder for neural program repair”, in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2021, pp. 341–353, ISBN: 9781450385626. [Online]. Available: `https://doi.org/10.1145/3468264.3468544`.